UiO : Department of Informatics
University of Oslo

Auto-tuning Shallow water simulations on GPUs

André B. Amundsen
Master's Thesis Spring 2014

Auto-tuning Shallow water simulations on GPUs

André B. Amundsen

15th May 2014

Abstract

Graphic processing units (GPUs) have gained popularity in scientific computing the recent years. This is because of the massive computing power they can provide for parallel tasks, and while GPUs are powerful, it is also hard to fully utilize their power. A part of this difficulty comes from the many parameters available, and tuning of these is necessary to maximize performance. Tuning consists of finding the best possible combination of those parameters. Separate tuning is needed even for GPUs from the same vendor and same hardware generation. Manually tuning these parameters is tedious and time consuming, and we therefore explore automatic tuning of such parameters. In this thesis we explore this problem for a shallow water simulator. We have successfully applied auto-tuning, making the program do the required tuning by itself. This has yielded an increase of 10-30% in performance, over manual tuning, on different NVIDIA GPU models. In addition we have implemented other mathematical approaches for solving the equations, and shown that which approach that is the fastest is different for different GPUs. A way of automatically selecting the best approach was also implemented.

Contents

List of Figures

Acknowledgements

I would like to thank my supervisors André R. Brodtkorb, Franz Fuchs and Martin Reimers for supporting me throughout the thesis. I would also like to tank SINTEF for providing me with an interesting topic, hardware, coffee and fruit.

Chapter 1

Introduction

During the last decade, graphics processing units (GPUs) have had a great increase in popularity for use in scientific programming. The reason for this increase in popularity is due to the GPUs focus on computational power. While they are more powerful they are also harder to program. The way they are programmed furthermore involves a lot of options that have a large impact on the performance of the compiled code. Selecting the best of these options is non-trivial and time consuming for all but the simplest kernels. To combat this issue it is beneficial to utilize a concept called auto-tuning. Auto-tuning is essentially to make the program itself find the combination of options that yield the best performance.

In this thesis, we explore the auto-tuning of an existing simulator[2] which uses NVIDIA GPUs. This shallow water simulator is a fitting target for GPU auto-tuning, as it is a program with many and complex kernels. Both the number of kernels and their complexity make this a non-trivial case where manual tuning is time consuming, and unlikely to yield optimal results. Any manual work will also be a weighting of effort versus improvement, and further limit the likelihood of optimal tuning. Some form of auto-tuning would save time and possibly yield better results[1, 4, 6, 9, 18, 25]. No longer requiring manual effort before running the simulator would also be an added benefit.

We will also look at alternative mathematical methods (schemes) for solving the shallow water equations at the core of this simulator. We do this as different approaches may be more efficient on some GPU models. Our complete auto-tuning approach therefore consists of both finding the best parameters for each single kernel as well as an automatic selection of the best scheme.

Our auto-tuning should, through better hardware utilization, provide more computational power. With better utilization and more power comes the possibility to run larger or higher resolution simulations, and could also make some simulations feasible on less costly hardware. Better power utilization also fits the resent movement towards *green computing*, which essentially is to make sure you waste as little power as possible doing your computations.

1.1 Auto-tuning of GPU code

With the above in mind our auto-tuning becomes two-fold. On one hand we want to create some form of dynamic tuning of parameters. We believe we can improve the speed of simulations on some architectures by providing different schemes that utilizes the hardware's resources in different ways. Selecting the fastest scheme presents a new issue that has to be handled.

On the other hand we need to find the most optimal number of threads for all our functions executed on the GPU, called *kernels*, and their variations. This number of threads, or *blocksize* which is it most commonly called, is the most important option we can tune. In the terms of auto-tuning this is often referred to as the *search space* in which we wish to find the most optimal combination of options. The problem is further complicated by the lack of continuity in the search space[21]. Due to templates with conditionals for different situations, we end up with 20 actual compiled flux and time integration kernels. Even if we limit the number of possible options by some heuristics we will still have a large search space.

To solve the above two issues this thesis will touch thee research topics. Our first topic relates to the complexity of tuning GPU based applications, and getting the most out of a specific scheme and its kernels, mainly with respect to block sizes. The kernels are large and complex, which makes this a non-trivial case. To achieve this we have implemented an empirical search using *micro-benchmarking* and *pruning* of the search space.

The second topic targets the possibility that the previously existing implementation of the current mathematical scheme may not be the most efficient for certain hardware. We have implemented two new variations of the existing scheme, where one is a purely programmatic change in the way fluxes are calculated, and the other is a mathematically different scheme. This other scheme is a dimensionally split version of the already implemented scheme.

The third topic is automatic selection of which scheme to use. We found that which of the schemes that is the most efficient varies across different hardware. To address this issue, we found a way of dynamically selecting the fastest scheme.

1.2 Related work

Auto-tuning in general is a topic where a large amount of work has been performed over the last decades. Auto-tuning of GPUs in particular has also had some attention the last decade. We will briefly describe some others' approaches to auto-tuning, and consider if any of them are suitable for our case. When considering these approaches, we will evaluate them on two fronts. These two fronts relate to the two-fold challenge we find auto-tuning our implementation to be. To find the optimal options within a search space a few categories of auto-tuning have been developed.

Empirical and Model driven auto-tuning Auto-tuning of code is roughly divided into two types, empirical and model driven [4–6, 15, 16, 18, 24].

Empirical tuning consists of timing every combination of options for every input, and remembering the fastest combination for each input. This has been seen to be quite time-consuming to run[6]. Two techniques are used to reduce the search space to save time, these are *pruning* and *micro-benchmarking*. Pruning entails searching only the parts of the search space that is found relevant using code specific limitations, metrics and/or characteristics[6, 21]. *Micro-benchmarking* saves time by only timing a few executions of the relevant parts of the code, as opposed timing full program executions[18].

The other type of auto-tuning is model based. This requires the programmer to find or create a model or simulator of the hardware memory hierarchy and other factors, such as in [18]. With GPU's hardware being very complex it is too difficult to model it accurately enough[15, 18]. We would also have to create a new model for each new generation of hardware that is released. Perhaps even one for each GPU model we wish to use within each generation. This is exactly the sort of manual labour we wish to eliminate.

Dynamic auto-tuning A very different approach is *Dynamic auto-tuning*, making the program automatically adapt to the current domain and hardware at runtime. This approach does not have the difficulties of searching a prohibitively large search space or creating models of complex hardware. It does however require the programmer to invent a heuristic for the choice of kernels and schemes.

Frigo and Johnson used one such approach with FFTW3 [8]. They describe FFTW as: '...discrete Fourier transform (DFT) that adapts to the hardware in order to maximize performance'. While their implementation does not utilize GPUs, they do have a selection of algorithms, and showed the algorithms' performance to vary with the CPUs they were run at, and with the dimensions of the input. They use micro-benchmarking to estimate the speed of their different solvers, and use these estimations to iteratively break down the problem into a plan for solving it.

Machine-learning Machine-learning (ML) for auto-tuning is something that has gained popularity the last two decades[1, 3]. ML is essentially a group of algorithms that are 'trained' using programmer specified data. Once trained, these algorithms can produce accurate predictions from inputs which were not a part of the training.

Training usually consists of generating data in advance, and use it as input to your choice of machine-learning algorithm. The algorithm would then generate a model. This model can later be used with a corresponding prediction algorithm for classification, or to predict run speeds (regression).

The generation of data normally consist of compiling kernels with a choice of different options and run each of these on a choice of inputs. The parameters of these kernels, the input, and the execution times would form

the basis of the training set.

Machine-learning in the field of auto-tuning can be used as the modelling in a model based auto-tuning approach[1]. It can also include more dynamic variables like program input parameters. There are several types of machine-learning algorithms available, amongst them support vector machines, neural networks and decision trees[3]. We will not go into those.

1.3 Application to shallow water equations

In this section we will look at how the methods used in related work applies to our case of shallow water simulation. Shallow water simulation features two main computations, the calculation of fluxes, and time integration. These are described further in Section 2.1. The functions calculating these on a GPU are called *kernels*, and in our case there are some variations of each. In total we have 20 kernels to tune in our auto-tuning case, each of which can have different optimal number of threads, called *blocksizes*. These 20 kernels are divided between 3 schemes, which we wish to find the fastest one of.

Each of our 20 kernels has between 24 and 2048 possible blocksizes. By applying *pruning* and *micro-benchmarking*, an empiric search may be possible to find the best blocksize for each kernel, but will require some running time after compilation. This is only possible because the efficiency of our kernels does not vary significantly with the input parameters.

Our case is similar to that of FFTW3[8] in that the performance of their schemes vary across hardware. We do however, not have the issue of it varying depending on input. While a sort of dynamic programming like in FFTW3 may help in selecting a scheme, it will not solve our problem of selecting blocksizes. Their approach is also unnecessarily complicated for our case, as our schemes performance will not vary within the context of a singe GPU. For our case it would be sufficient to do the selection of a scheme once per GPU, instead of doing a selection upon each program execution.

Brodtkorb et al.[2] have already implemented a form of dynamic programming. They have an automatic switching between kernels that can avoid computation of dry areas, known as early-exit kernels, and normal kernels which always does computation of the full domain. The dynamic switching is used to make sure the most efficient type of kernel is always used. We have incorporated this early-exit solution into our auto-tuning.

We did also consider a machine-learning approach, but concluded that such an approach would not add anything of value, but be more complicated to implement. This is because the optimal performance is often found on 'cliffs' in the search space that are created by hardware constraints such as the size of shared memory or number of threads, and tipping over this cliff may yield a very slow kernel, or even one that will not compile. For a machine-learning algorithm to be accurate the learning data would have to contain information on these cliffs. Finding these cliffs

would require a modelling of the GPU resources or a full empirical scan of the search space. As described earlier modelling is difficult, and if we are doing a full empirical scan there is nothing machine-learning would gain over a straight forward numerical comparison of execution times as a way to find the best performing blocksize.

With machine-learning it could be possible to avoid creating a new model for each GPU by also using GPU information as part of the training input and using training data from multiple GPUs to create one model. We could then distribute the model along with the code instead of learning anew on each GPU. This thesis does not cover this subject but suggests it as a topic for future research.

For this thesis we have used empirical search with pruning and micro-benchmarking to find the best blocksizes. To select the best scheme we employed a simple dynamic programming concept.

Thesis structure In order to produce some form of auto-tuning we need to understand the shallow water equations and how they can be used. The next chapter will describe the mathematics and algorithms of the shallow water equations and the numerical scheme used to solve these in the simulator. After the mathematics has been presented we will present NVIDIA CUDA and how the initial implementation of the above scheme efficiently utilizes GPUs. We will then continue with a description of the new kernels and schemes. Following that will be a description of our auto-tuning approach. In the end we will show data on the effect of our implemented auto-tuning approach.

Chapter 2

Background

Before we describe our auto-tuning approach we need to present some background knowledge of the mathematics and programming concepts involved. This chapter will provide those by introducing the mathematical scheme, the CUDA programming model, and how the current implementation utilizes CUDA to solve the mathematical scheme.

2.1 The shallow water equations

Here we present the equations for shallow water flow, and the numerical scheme used to approximate the solution of these. These are the equations and scheme Brodtkorb et al.[2] have implemented on NVIDIA GPUs, and we base our work on.

The shallow water equations, are limited to environments where the vertical movement is negligible compared to the horizontal movement [10]. Even with these restrictions it applies in many cases, like dam breaches, tsunamis, and river flows. These equations are conservation laws, which means the total water volume and total momentum is constant over time. The quantities can of course be distributed and change at a given point in the domain, but the total over the domain will stay the same. The shallow water equations with the bed slope source term in two dimensions are

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}g\,h^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -ghB_x \\ -ghB_y \end{bmatrix} \tag{2.1}$$

The variables are explained in Table 2.1. There is also an added bed shear stress friction term, which can be found in [2].

To simplify later equations we also present (2.1) in vector form

$$Q_t + F(Q) + G(Q) = H_B(Q, \nabla B), \tag{2.2}$$

where $Q = (h, hv, hu)$ is the vector of conserved quantities, F and G are the fluxes in x and y direction respectively, and H_B is the bed slope source term.

The shallow water equations (2.1) cannot directly be used for simulation, as they have no known mathematical solution in the general case. Therefore, schemes have been developed to approximate the solution. One of these schemes is the Kurganov-Petrova scheme[13], which is implemented in the simulator we base our work on. The Kurganov-Petrova scheme is a central-upwind scheme of the Godunov type[13], and is a second order scheme used for approximating the solution of the shallow water equations. The approximation is done by discretizing the shallow water equations in both space and time, which is achieved by introducing *time-steps*, n, and a grid of cells, each with a midpoint-average of Q in that cell. Also, as a prerequisite to make the scheme well-balanced, Kurganov-Petrova switches from the physical variables, (h, hv, hu), to the derived variables (w, hv, hu), in which w is the water surface elevation (see [13] for details). Q will therefore denote the derived variables for the remainder of this thesis. The transformation of h can be done using $h = w - B$.

2.1.1 Discretization in space

As stated above, the discretization in space is done by describing the domain using a grid of cells with the average of Q for each cell. By solving (2.2) for Q and discretizing we get

$$\begin{aligned}\frac{dQ_{ij}}{dt} &= H_B(Q_{ij}, \nabla B) \\ &\quad - \left[F(Q_{i+1/2,j}) - F(Q_{i-1/2,j})\right] - \left[G(Q_{i,j+1/2}) - G(Q_{i,j-1/2})\right].\end{aligned} \tag{2.3}$$

Figure 2.1 shows a cell and the coordinates i and j. From this figure and the formula we can see that the fluxes are calculated at four points reconstructed from the centre-point value. For these reconstructions the Kurganov-Petrova scheme uses the generalized minmod flux limiter[11]. Unfortunately this reconstruction may cause negative h in shoal zones. Negative h will cause problems, as the eigenvalues for the shallow water equations are $u \pm \sqrt{gh}$. To address the possible negative values of h from this reconstruction, the slope of w is altered such that the value of the reconstructed h at integration points will be non-negative. With a non-negative average, the values of four integration points can be guaranteed

Variable	Description
Q	The quantities involved. For our use it incorporates h, hu and hv.
h	Water depth
hu/hv	Water momentum in x and y direction respectively
g	Gravitational constant
n	Denotes time-step
F/G	Fluxes in x and y directions respectively. Approximation of the physical force
B	Bathymetry. The elevation of the water bed.

Table 2.1: The variables of the shallow water equations.

to be non-negative because we also have bilinear bottom topography and a planar water surface. This limits us to second-order accuracy[12]. Altering the slopes of w will cause some minor waves to form for shoal zones, but should be negligible to the solution.

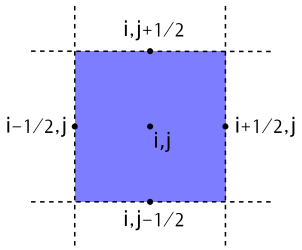


Figure 2.1: A cell with reconstruction points and the coordinates i and j.

The bathymetry, B, is given at cell corners and reconstructed for the cell edges using a piecewise bilinear function. The fluxes are calculated using the central-upwind flux function[12], where a step is to find u using $u = hu/h$. This calculation will cause large round off errors as h approaches zero, and thus can cause very slow propagation of the solution as our maximum timestep-calculation is based on these velocities. To handle this, the calculation of h is *desingularized* when $h < \kappa$. This *desingularization* makes the scheme well behaved for *shoal zones*[12]. Choosing a value for κ, however, is not simple. Large values will produce large errors, while small values will yield very small timesteps and thus slow down the simulation. Kurganov and Petrova's approach is not suited for many real-world applications[2], therefore we instead use a κ linearly proportional with the grid resolution. See [2] for details.

2.1.2 Discretization in time

So far we have described a semi-discrete scheme. We now need to discretize with regards to time. As mentioned earlier in this chapter, this is done by introducing *time-steps*, n.

Evolving these discrete values in time, i.e. going from *time-step* n to *time-step* $n+1$, can be done in numerous ways. We use the Runge-Kutta ordinary differential equation solver[22]. In conservative form it is written as

$$\begin{aligned} Q_{ij}^{n+\frac{1}{2}} &= Q_{ij}^{n} + \Delta t R(Q^n)_{ij}, \\ Q_{ij}^{n+1} &= \frac{1}{2} Q_{ij}^{n} + \frac{1}{2}\left[Q_{ij}^{n+\frac{1}{2}} + \Delta t R(Q^{n+\frac{1}{2}})_{ij} \right]. \end{aligned} \tag{2.4}$$

The above equation, (2.4), describes the change of the quantities, Q, from time-step n to time-step $n+1$ according to the calculated Runge-Kutta flux, $R(Q)$, for time-step n and time-step $n+\frac{1}{2}$. Q_{ij}^{n} is shorthand notation for the averaged quantities, $\overline{Q}$, at cell i, j at time-step n, or in a different notation $\overline{Q}(x_i, y_j, t^n)$. Notice Δt being the same one for the $n+\frac{1}{2}$ step as well as the $n+1$ step.

The CFL condition is a necessary requirement for the scheme to be numerically stable[2, 14]. In our case it reads as,

$$\Delta t \leq \frac{1}{4}\min\left\{\Delta x/\max_\Omega\left|u \pm \sqrt{gh}\right|, \Delta y/\max_\Omega\left|v \pm \sqrt{gh}\right|\right\} = \frac{r}{4}. \quad (2.5)$$

We use this formula together with approximate eigenvalues to choose a timestep that satisfies this condition.

Our implementation also uses a more accurate description of bed shear stress than that of Kurganov and Petrova's original scheme to satisfy requirements of real-life cases. For details we refer the reader to [2].

Our implementation can also run first-order Euler scheme steps. For an Euler scheme we simply use only the first line in 2.4, and treat $Q_{ij}^{n+\frac{1}{2}}$ as a whole timestep.

The equations presented so far shows that each cell can be calculated separately, which creates a high potential for parallelism. Parallelism is important to efficiently utilize GPUs.

2.2 CUDA

To fully exploit the power of a GPU one needs to understand its architecture. This section will provide the background needed by describing the architecture of NVIDIA graphics cards, and how the simulator utilizes it. Firstly we will briefly explain the basic differences between CPUs and GPUs. Then we will describe what CUDA is, and the main principles of NVIDIA architecture. Lastly we will present how our implementation utilizes said architecture.

SIMD computing Graphics cards, including NVIDIAs, gain power by sacrificing versatility. Instead of the complex control logic that is found in CPUs, GPUs have only a small part of its transistors dedicated to controls and thus have a higher number of transistors doing actual computation. A different paradigm is needed to fully utilize such an amount of transistors with little control logic. Instead of the traditional single instruction single data (SISD) used by CPUs, GPUs use single instruction multiple data (SIMD)[1]. In other words they do the same operation on many points of data at the same time. Programming using this paradigm is different and requires some extra notation or syntax compared to traditional programming languages. This brings us to CUDA, as NVIDIA GPUs are our hardware of choice.

CUDA overview NVIDIA has developed their own language for programming their graphic cards. This language is CUDA, and is strongly based on C++, but has some restriction and additions[19]. A separate compiler, *nvcc*, is provided to handle CUDA. This compiler handles any code

[1]SISD/SIMD definitions by Flynn[7].

containing CUDA specifics, compiles it to a normal linkable, compiles normal C++ code through g++ or other compilers and then finally links everything using standard linkers.

NVIDIA also provides two APIs to interface with the cards; the driver API and the runtime API. The current implementation uses the runtime API, which is the easiest to use and most common. This API provides functions for memory allocation, data copy between the card and CPU, and *kernel* handling. In CUDA jargon, a *kernel* is simply a function which is intended to be executed on the GPU. The CUDA toolkit[2] also comes with a visual profiler and CUPTI, a runtime profiler, which was used in this thesis to gather data on the kernels performance.

2.2.1 Architecture and terminology

To use this CUDA toolkit properly, an understanding of CUDA programming concepts is required. It is as stated earlier a SIMD programming environment which demands some extra care and thinking.

One of these concepts is how *threads* are grouped and work together in a SIMD fashion. *Threads* are much the same as in a CPU, with one difference. In a CPU each thread has its own instruction pointer and gets its own instructions at its own pace. In NVIDIA GPUs the threads are gathered into groups of 32 called *warps*. All threads in a *warp* are given the same instructions at the same time. This makes it important for the programmer to keep threads within a warp at the same execution path. If execution paths diverge, the instruction dispatcher has to use additional cycles for each divergent path[3], and results in threads idling while waiting for instructions. This allows for a worst case scenario of 32 divergent threads, which will reduce performance by a factor of 32; hence the importance of eliminating as much branching within each warp as possible.

Furthermore, only threads running in the same *block* are grouped into warps. A block is simply a grouping of threads that the programmer intends to work in tandem, and it is through these blocks a programmer specifies how many threads to use. The number of threads and blocks are chosen through code like in the minimal example below.

```
dim3 grid = dim3(5,5,1); //We want to run 5x5 blocks

//Define block dimensions of 16x16 threads
dim3 blockDim = dim3(16,16,1);

//Launch a kernel with the dimensions above
my_kernel<<<grid, blockDim>>>();
```

As you can see, blocks are further grouped in a *grid*. This is further illustrated in Figure 2.2. A grid is the grouping of all blocks running a

[2]The CUDA toolkit version used for this thesis is 5.5

[3]Technically all 32 threads are scheduled the same instructions, but the threads that don't need them will be filtered out using a mask.

single invocation of a kernel.

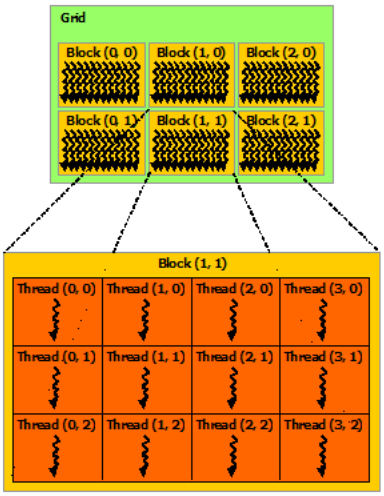


Figure 2.2: This figure illustrates the CUDA thread hierarchy. Threads are grouped into blocks which are further group into grids. Figure by NVIDIA[19].

These partitioning terms are required to keep in mind when writing kernels. When writing a kernel it should be thought of through the scope of one block. Within a block the programmer has access to several important variables. The most important ones are blockIdx.x/blockIdx.y and threadIdx.x/threadIdx.y. These are used together with blockDim.x/blockDim.y to calculate which part of the domain the current block should work on, and which element of that part each thread should work on. This is how one can divide the threads to work on different parts of data.

These levels of grouping also have access to different levels of memory. As with most processors, the NVIDIA cards also have a memory hierarchy. Starting with the smallest and fastest, each thread has its own registers and local storage, just like CPU threads. Higher up the chain we have the *shared memory*. Shared memory is about the same level as a L1 cache for regular CPUs, and is accessible by all threads within a block. This memory is used extensively in the simulator to minimize *global reads*. *Global reads* refers to reading of data from the *global memory*. *Global memory* is the DRAM, the largest memory available on a GPU, and is the one manufacturers advertise in product descriptions. Throughout literature it is also referred to as *device memory*. The most important types of memory, registers, shared and global, are illustrated in Figure 2.3. In addition to these, we have *texture memory*, *surface memory* and *constant memory*. Constant memory is a small part of the global memory that is optimized for caching, and is read-only by threads; hence the name. The last two, texture and surface memories, are extra caches optimized for graphics rendering. We also have the usual computer RAM, *host memory*, which is not directly accessible from GPU threads. Data first have to be moved from host to device by use of the API.

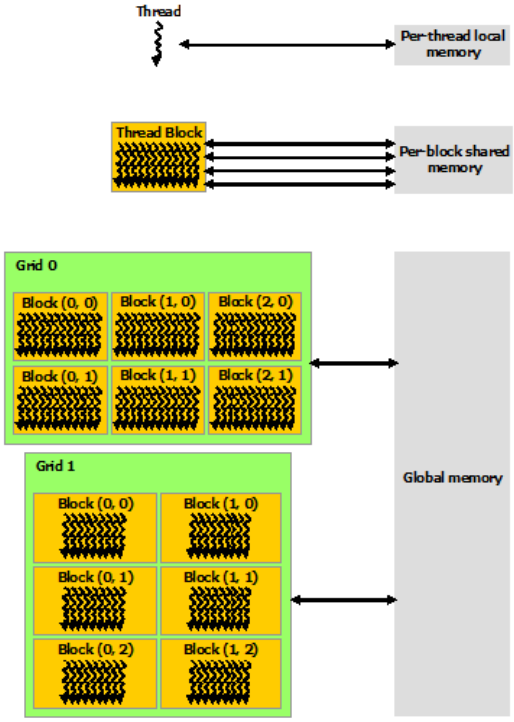


Figure 2.3: Illustrates the CUDA memory hierarchy. Each threads has its own separate registers, each can communicate through shared memory, and all threads share access to global memory. Figure by NVIDIA[19].

2.3 Utilizing the architecture

In this section we will describe the structure of our implementation and how it utilizes the hardware described in the last section. The kernels implemented in [2] will also be described.

The controlling part of the simulator is a C++ class. This class handles initialization, kernel set-up and launches. It also functions like an API for other C++ applications using the simulator[4]. This manager class is not particularly interesting, it is the kernels it manages and the data layout that are.

Before we describe the kernels we have to explain how the data is organized. While object oriented programmers may be tempted to create a class 'cell' and have it contain Q, this is not efficient. Each of our variables, w, hu, and hv are contained in separate arrays, U1, U2 and U3 respectively. We also have another set of arrays, Q1, Q2 and Q3, which are used for $Q^{n+\frac{1}{2}}$ in equation 2.4. Having each variable in a packed memory space allows the NVIDIA hardware to utilize its large size reads, and allows for caching in the unused parts of shared memory. This also maximizes bandwidth for device-to-host and host-to-device memory transfers. To

[4]The two main programs using the simulator are *kp*, and *kp_visualization*. *Kp* works entirely through the command line, and is capable of printing to file in NetCDF format. *Kp_visualization* runs a 3d visualization of the simulation.

simplify the kernels and avoid some conditionals we also pad the domain to be a multiple of the chosen blocksize. How the domain is split into blocks and padded can be seen in Figure 2.4. In this Figure we can see how the computation can be done for one cell at a time, it only depends on the values of the cells around it for the previous timestep. Sometimes cells that are not going to be processed themselves, are needed for the computations of other cells. This happens for all cells whose stencil is partially outside the block. The extra cells are called *ghost cells*, and are located outside the domain or in other blocks than the current one.

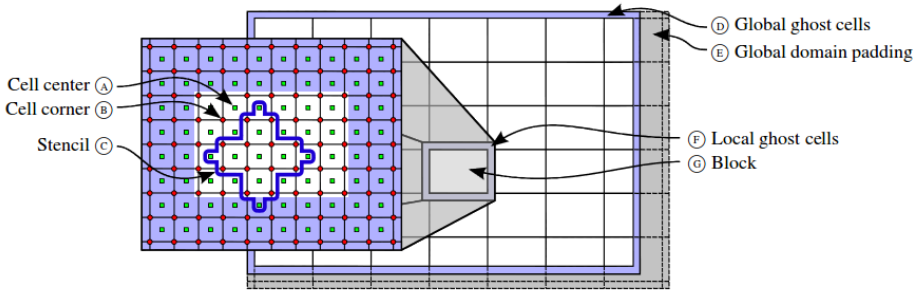


Figure 2.4: Depicts the domain, its ghost cells, and the stencil used for the calculations. Figure by Brodtkorb[2].

While a packed memory layout is important, one also have to access it in efficient patterns. This brings us to the workings of the kernels. The simulator has four types of kernels, here in order of computational demand: Flux-calculation kernels, time-integration kernels, dt-kernels and boundary-condition kernels. There is only one sort of each kernel type in the original code, but this will be expanded upon to create more opportunities for maximal utilization of the architecture.

These will be presented in the following chapter. The call order of the kernels can be seen in Figure 2.5. First the fluxes are calculated, then the dt-kernel finds maximum timestep allowed by the eigenvalues found during flux calculation. Then the time integration kernel evolves the solution in time using the fluxes found by the flux calculation kernels. Lastly the boundary conditions are updated. The flux calculation, time integration and boundary updates are repeated in the case of the Runge-Kutta 2 scheme to achieve second order accuracy in space.

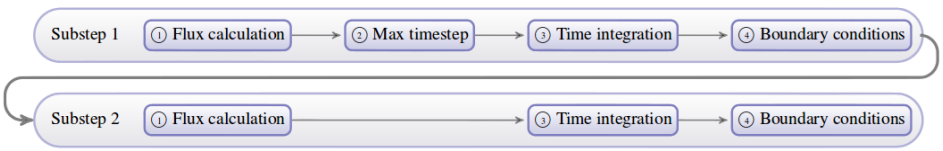


Figure 2.5: Describes the flow of kernels calls. Substep two is only used by the second step of Runge-Kutta 2. Figure by Brodtkorb[2].

Flux-kernels The flux-calculation kernels are the kernels that find R in 2.4. These are the most computationally heavy kernels. In order to avoid many slow global memory reads, these kernels heavily utilize shared memory. By first reading the conserved variables and the bathymetry from global memory into shared memory we eliminate repeated slow global memory access to the same value. Instead we have repeated access to the much faster shared memory. The reading from global memory is done by *striding*. Striding is essentially a double for-loop where $blockDim.x \times blockDim.y$ values are read at a time, iterating through the data needed in a block-wise fashion until the required data is read. Striding is needed since the ghost cells makes a block use more values than it has threads. This ensures that consecutive threads reads consecutive data, and so minimizes the number of global reads by utilizing the architectures ability to join many small consecutive reads into a larger one. Refer to Figure 2.6 for a brief overview of the calculations done in the flux kernel.

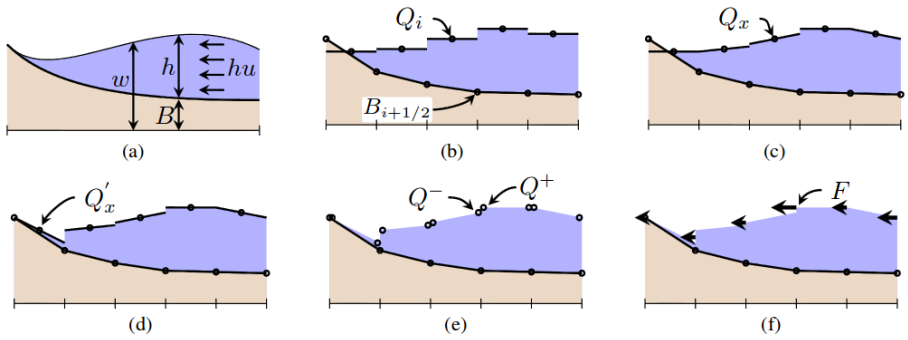


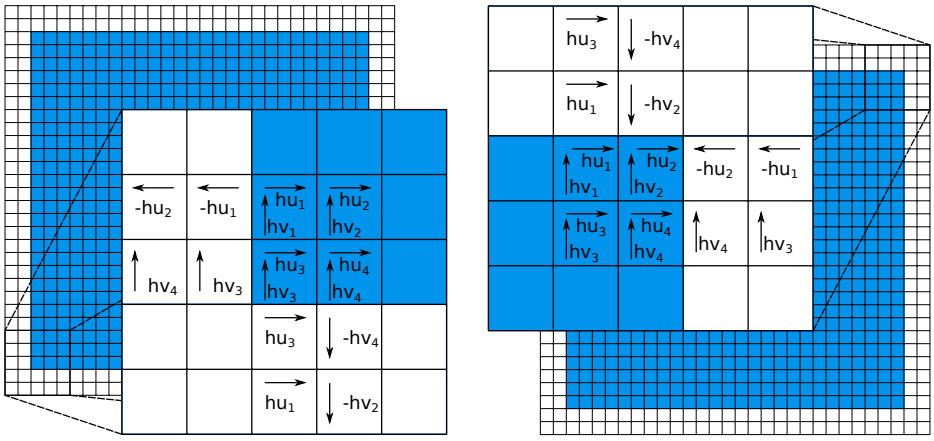
Figure 2.6: The stages of flux calculation. (a) The variables involved; (b) Q is given as cell-centre averages. B is given at cell intersections, and reconstructed using a piecewise bi-linear function; (c) Reconstruction of Q using a general minmod flux limiter; (d) Surface modification of wet-dry interfaces to avoid negative water depth; (e) Reconstruction of point values at integration points; (f) Fluxes computed using the central-upwind flux function. Figure by Brodtkorb[2].

Once flux calculation is done, the results are written to a separate memory space, R, for use by the time-integration kernels. r from equation 2.5 is also calculated for each interface for the first stage of Runge-Kutta 2, and the smallest one is found by *shared memory reduction*. *Shared memory reduction* is a technique to find the larges/smallest value within a block. In a loop the threads compare two values each and store the desired one at the smallest index. Each iteration of the loop the number of threads is halved, and thus the number of values to compare each iteration is halved. In the end only one value is left.

Dt kernel The Dt-kernel also uses shared memory reduction. It simply reads through r using this shared memory reduction and finds the smallest value. This r is then used to set the timestep using equation 2.5.

Time integration kernels The time integration kernel reads the required variables, Q, B and R. Since there is no need for ghost cells to do time integration, the variables are simply read into each thread's registers. The kernel then reconstructs h as per $h = w - B$, and calculates manning friction. Before evolving the solution in time the kernel may do a shared-memory reduction to check for wetness. This is used for an early exit optimization which is explained later in this chapter.

Boundary condition kernels The final type of kernels is the boundary condition(BC) kernels. The BC kernels consist of many different types of kernels, each modelling a type of boundary condition. Only the boundary condition *wall* has been used for this thesis. This BC models an ideal reflective wall. It does so by copying our conserved variables, w, hu and hv, from cells inside the domain to the ghost cells. Each two outermost cells inside the domain are copied to their two corresponding ghost cells. See Figure 2.7. Furthermore, the direction of hu needs to be reversed for west and east boundaries, and similar for hv and north/south boundaries. By having Q mirrored over the boundary the forces will equalize and the boundary behave like a wall.



(a) South-West domain corner.

(b) North-East domain corner.

Figure 2.7: Copying of values for boundary condition wall. The two cells closest to the boundary have their values copied to its corresponding cell outside the computational domain. The ghost cells show only one value each for readability.

As mentioned, there is an early exit optimization implemented. It is fairly simple and consists of reading and writing to a wetness-map. Each block of the time integration kernels checks for wet cells, and writes zero or one to the field corresponding with block-id. When the flux-kernel then is run, each block checks itself and it's closest 4 neighbours if they are wet. If none are, the block simply returns without further operations. The neighbours have to be checked as there could be water in the local ghost cells, causing the current block to become wet this timestep. This

implementation requires that the flux kernels and the time integration kernels have the same blocksize, so that the areas checked by the time integration kernels correspond to the areas computed by the flux kernels.

Chapter 3

Alternate numerical schemes

Different GPUs have different resource limitations. For lower-end cards the limitation is often floating point operations, while for high-end cards it often is the bandwidth from the processors to the device's memory which is the limiting factor. As a result the resource utilization of a single algorithm may vary from GPU to GPU. Other ways of implementing our numerical scheme, or different numerical schemes, can therefore provide faster execution on some architectures. This chapter will describe a variation on the current numerical scheme, and a numerically different scheme, for approximating the solution of the shallow water equations. First we will describe how these new schemes work and why we think they provide opportunities for faster execution. Afterwards we will present verification of correctness and a comparison of the speed of these schemes.

3.1 Implementation

We have created two different implementations. Firstly, we created an alternate way to calculate the fluxes. The scheme is still the same as for the original implementation(Kurganov-Petrova), but we calculate the F-fluxes and G-fluxes in separate kernels. Secondly, we implemented a dimensional split version of Kurganov-Petrova. This dimensional split scheme was also extended to second order accuracy in time.

3.1.1 Separate kernels for flux calculation

In the first approach the calculation of the fluxes(2.3) was split into two different kernels, one calculating the F-fluxes, and the other calculating the G-fluxes. These two kernels will throughout the rest of the thesis be referenced as Fx-kernels and Y-block kernels respectively. This will reduce the number of registers used, and reduce shared memory usage per block, but will increase the number of reads from global memory. The Fx-kernel was also improved to only have one dimension to best utilize the hardware through long coalesced reads and writes. The Y-block kernel is the same as the original flux-kernel, except calculations of F-fluxes and corresponding eigenvalues were removed. Since this kernel only calculates Y-fluxes, ghost

cells are only needed in the Y-direction, and all reads from global memory could be simplified to single for-loops. This kernel was not changed to only one dimension as global reads in Y-direction would be un-coalesced and slow. Having two kernels that calculate fluxes at the same timestep requires either one additional float field per conserved variable or requires the flux kernel executed last to do a very slow read-then-write. We have added an additional float field.

A new variation of the Runge-Kutta kernels was also created, as the fluxes now have to be read from two arrays per conserved variable, instead of one. These will be referenced as SRK2-kernels (Split Runge-Kutta 2). This implementation has simpler kernels that use less shared memory, but requires more reading from global memory, and so should benefit GPUs where the computations are the limiting part, as opposed to the memory transfers.

3.1.2 Mixed order dimensional split

Building upon the kernels from the previous section, the next step was to create a single-line kernel calculating the G-fluxes as efficient as the X-kernel. In order to accomplish that in a way that allows for fast global memory reads, we needed to transpose the domain. Otherwise, the GPU would be wasting most of its read/write bandwidth on single unpacked float values. If we were to do this using the existing numerical scheme, we would have two alternatives. The domain could transposed between the X- and Y-kernels and back again each *substep*. Alternatively, we could have two sets of the domain, one transposed and one normal. Both these options greatly increase the number of required reads and writes to global memory and so are quite inefficient. Instead we decided to go with a dimensional split(DS) scheme. For a second order DS scheme the domain would only have to be transposed two times per two timesteps. The reasoning for this will be presented in the next section along the second order DS scheme. In this section we will describe our mixed order version of the DS scheme and its implementation. The scheme is mixed order as the flux calculation is second order and the time integration is first order. The next section will then expand this to fully second order.

A general first order dimensional split scheme reads as follows,

$$U^{n+1} = X^{(\Delta t)} Y^{(\Delta t)}, \tag{3.1}$$

where X and Y are substeps with flux calculation in one direction and time integration[23]. In our scheme X and Y are Euler timesteps with fluxes in only one direction,

$$\begin{aligned} X^{\Delta t} \quad & \left(\tilde{U}^{n+1} = \tilde{U}^n + \Delta t F(\tilde{U}^n)\right) \\ Y^{\Delta t} \quad & \left(\tilde{U}^{n+1} = \tilde{U}^n + \Delta t G(\tilde{U}^n)\right), \end{aligned} \tag{3.2}$$

where the fluxes F and G are the same as in (2.3), and $\tilde{U}$ is the conserved variables within the context of the current sub-step. The Equations (3.1) and (3.2) together make up our mixed order DS scheme.

Transposing the domain With this mixed order DS scheme we still have to transpose the domain between the *X*-sub-step and the *Y*-sub-step. This scheme does however present us with a possibility of accomplishing the transposition without extra reading and writing to global memory. This possibility consists of doing the transposition by modifying the time integration kernels to output in a transposed manner instead of the normal order.

Sadly, straight forward transposition in the time integration kernel by simply swapping coordinates is not effective. A part of this is due to the GPUs global memory being divided into partitions that can operate independently, but where all read/write request targeting the same partition gets serialized. Is known as *partition camping* if a kernel has a read/write pattern which cause this serialization to happen often. Transposition by simply swapping coordinates causes partition camping as all the thread blocks of the same row will write to the same partition when outputting, effectively limiting the global memory bandwidth to $total\_bandwidth/n\_banks$. To avoid this we did transposition in shared memory, with padding to avoid shared memory bank conflicts, and applied the diagonal reordering from [20]. Even with the application of diagonal reordering, transposing the domain increases execution times. In our case the execution times of the time integration kernels increased by approximately 30%. A part of this loss of performance comes from a few extra computations to translate from diagonal order to Cartesian order, and from a less effective blocksize as the kernel now is restricted to a square size.

The transposition also creates an issue with regards to the bathymetry. The bathymetry, *Bi*, is not transposed as that would require more memory. The Y-direction flux calculation does however require 2 columns of *Bi*, which will cause un-coalesced reads due to *Bi* not being transposed. To somewhat limit this un-coalesced read problem, both cells of $blockDim/2$ rows are read at a time, halving the number, and doubling the size of reads. For details of how this is done, refer the code below.

```
int row = threadIdx.x % 2;
for (int i=threadIdx.x/2; i<width; i+=blockDim.x/2) {
  float *Bi_ptr = device_address2D(Bi.ptr,
                        Bi.pitch, by, bx+i);
  Bi[row][i] = Bi_ptr[row];
}
```

Time integration also requires bathymetry values. Since transposition of *U* is already being done by these kernels, the same shared memory, and the techniques from [20] mentioned above, are reused for reading bathymetry values as well.

Selecting timestep Our dimensional split scheme also differs from the RK 2 scheme in that it is harder to find a suitable time-step. Notice that the two substeps in (3.1) use the same Δt. To keep the scheme stable it is also required that this Δt satisfies the CFL-condition in both the X-substep

and the Y-substep. This is problematic since the eigenvalues, which are a component of the CFL-condition, can not be found in advance of both substeps. The simplest approach to solve this would be to select a Δt so small it is sure not to exceed the CFL-condition, but such an approach would cause a large unnecessary increase in computation time due to more and smaller timesteps. Instead we approximate Δt using

$$min(r\Delta t_x^{n-1}, r\Delta t_y^{n-1}), \tag{3.3}$$

where Δt_x^{n-1} is the largest Δt which satisfied the CFL-condition for the previous X-substep, and Δt_y^{n-1} is the equivalent with respect to the previous Y-substep. A scaling factor, r, is included to increase the chances our approximation has to be valid for the current timestep. By Formula (3.3), we select Δt based on the direction that was the most limiting in the previous timestep. Scaling down the most limiting Δt works for most timesteps as the fluxes, and through them the eigenvalues and the CFL-condition that limit Δt are unlikely to vary much from one step to the next. If the velocities increase to such a degree that the CFL-condition is exceeded, the scheme will become unstable. To handle this we calculate Δt in each substep and compare it to $\Delta t_{selected}$, which was selected with Formula (3.3). If $\Delta t_{selected} > \Delta t$ any calculations done up to this point in this timestep are discarded. Due to the transposition in the domain, the time integration kernel in the X-substep writes to a separate memory space instead of overwriting the initial data, so that no backup in necessary to redo calculations for the current timestep. The calculations are then attempted again with Δt using Δt_x^n and Δt_y^n since they both are known. The timestep for the Y-substep, Δt_y^n, was based on the erroneous domain after time integration of X-direction with a too large Δt, and is therefore scaled by r through Formula (3.3) again. The next attempt may still fail if this reduction was not enough, in which case the process of recalculating the timestep is repeated until a small enough Δt is found.

Since we are basing Δt on Δt_x and Δt_y from the previous timestep we still have the problem of setting an initial Δt. To find this we simply run the first timestep using the original Euler scheme.

Setting r, the Δt scaling factor in Formula (3.3), is not trivial. A small r will increase execution time through smaller timesteps. If r is too large it will cause many timestep resets due to exceeding the CFL-condition. We set up a single test case modelling a circular dam break to test the effect of changing r. The circular dam break case, shown in Figure 3.1, has a flat bathymetry, $B_{x,y} = 0$, a cylinder of water of $h = 2.5m$ and $radius = 2.5m$, in the middle of a domain with $length = width = 40m$. For this case the rest of the domain has $h = 0$, and simulation was run until $t = 2$. This is a standard test case described in [23]. For the boundary conditions we used wall, which was further described in section 2.3.

Figure 3.2 shows the effect of changing r, the Δt scaling factor, for the dimensional split schemes simulating case $cdam(500, t = 2)$. We can see there is a significant performance difference (up to 20%) depending on the choice of r. Based on this, r was set to 0.95, which this figure indicates is

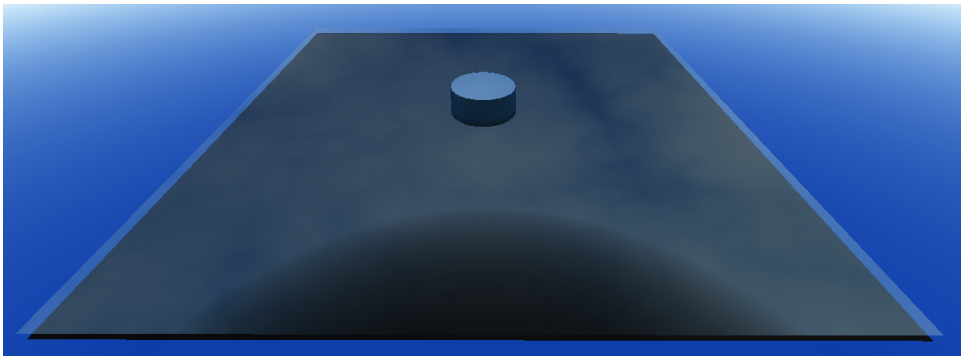


Figure 3.1: Circular dam initial state, with wet surroundings, $cdam(800, t = 0)$. The cylinder is centred on the centre of the domain, and has $R = 2.5m$. The domain is $40m$ x $40m$. The cylinder has $h = 2.5$, while the rest of the domain has $h = 0.5m$.

the value for r where the best balance between smaller timesteps and more recalculations is found. We have not explored this further, but note that it could be a topic for future research.

As it is the X-substep which sets Δt, we have an opportunity to use the actual Δt calculated in the X-substep instead of an estimate. By modifying Formula (3.3) to use the actual Δt for the X-substep we get $min(\Delta t_x^n, r\Delta t_y^{n-1})$. This should not cause instabilities as we still limit ourselves to the smallest value. By using that formula we would however get a more correct result if the X-direction fluxes are the limiting factor. In the case where the Y-direction is the more limiting factor, we would still use $r\Delta t_{n-1}^y$.

Boundary conditions The domain's *boundary conditions* (BC) are also affected by transposing the domain, because the transposition swaps the axis of the water momentum, *hu* and *hv*. Figure 3.3 illustrates this. The change of directions of the variables also has to be reflected in the BC kernels. This is done by reversing *hu* for north/south and *hv* for west/east, instead of *hu* west/east and *hv* for north/south as in the original kernels. To avoid creating an entire new set of BC-kernels we simply swap the arguments *Q2* and *Q3* (*hu* and *hv*) for the BC-kernels. This works for the BC-Wall kernel because the only difference between the handling of the two arguments, is whether the direction is reversed or not.

3.1.3 Second-order dimensional split

The last of the new schemes is the second-order dimensional split. This is surprisingly similar to the first order version, but instead of doing one step like in Figure 3.1, we do a double-step. For this double-step to meet second order accuracy, the two directions need to be in opposite order in the last step compared to the first. Mathematically this reads[23]

$$U^{n+2} = X^{(\Delta t)}Y^{(\Delta t)}Y^{(\Delta t)}X^{(\Delta t)}. \tag{3.4}$$

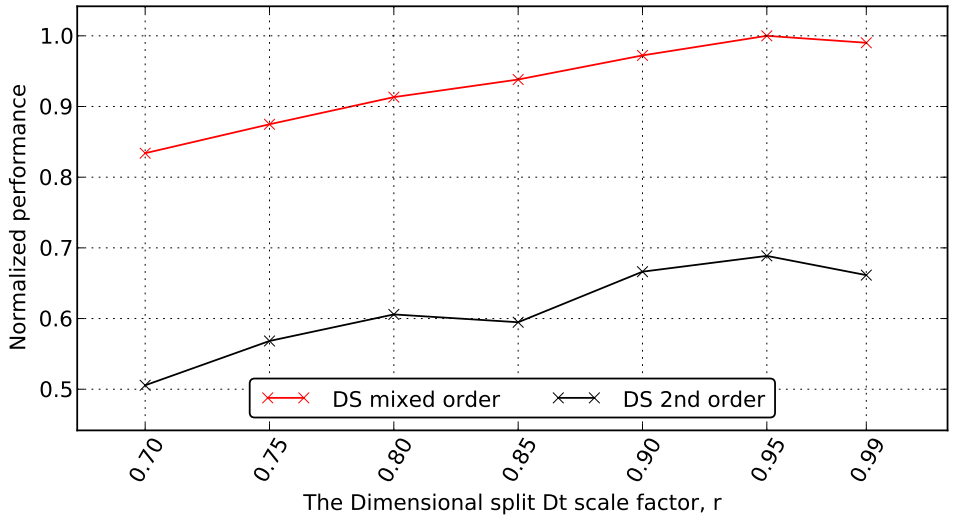


Figure 3.2: Variations in performance of dimensional split schemes with varying scaling of Δt. The case run is a circular dam break, cdam(500, t=2) with dry surroundings. This case has large variations in fluxes and is therefore heavily affected by variations in Δt. We can see an increase in performance up to $r = 0.95$, and then a small decrease to $r = 0.99$. From this we can say that at $r > 0.95$ the time taken by recalculating steps exceeds the time taken to calculate more and smaller steps.

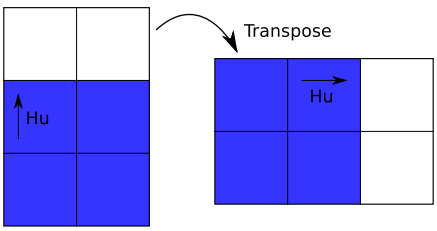


Figure 3.3: Transposing the domain also switches the direction of hu and hv.

In addition to alternating which direction is calculated first, each of the substeps also has to be second order accurate by themselves[23]. This second order accuracy is achieved by using Runge-Kutta 2 time integration for each sub-step as per equation 2.4.

Unlike with the first-order scheme, we do not have to transpose the data for each substep, but only between calculation of different directions, which is after the first X-direction step, and after the second Y-direction step. This is because we do two Y-steps in a row and both require the data to be in transposed order. Transposition is not needed after the last X-direction step either, as the data is already in original order. With these points in mind we should expect the execution time of the second-order scheme to actually be slightly less than double of the first-order scheme. While the number of calculations is doubled to reach second order accuracy, there are fewer performance reducing transpositions per

timestep. To accommodate both the transpositions and the second order time integration a new float field is required. Instead, to save memory space, we reused the extra float field introduced with the split flux kernels scheme. Only one scheme is run per timestep, so that such extra float fields can be re-purposed for use in other schemes.

Like with the first order dimensional split scheme, the second order dimensional split scheme also has the issue of selecting Δt. We have used the same concept as for the mixed order scheme, where we use Formula (3.3). We also save the input quantities, including domain ghost cells, for the current double-step in a new backup float field. This has to be done since the input fields are reused for calculations throughout the double step. If any of the four sub-steps find Δt to be exceeding the CFL-condition, the step will be restarted and input values copied from the backup field. Padding of the backup field is not needed since no calculations will be done directly on it, so we save some memory space by not padding it.

The selection of Δt is more complicated for the second order version, compared to the mixed order version because we now have to find a value that satisfies the CFL-condition for *two* timesteps (four sub-steps) in advance. In the case of Δt exceeding the CFL-condition, manual experimentation found that simply recalculating using (3.3) with the updated Δt_x and Δt_y in most cases caused many resets of the timestep. The problem was greatly reduced by halving Δt_x and Δt_y when resetting the timestep, before solving (3.3) again.

This dimensional split scheme has long kernels with only one dimension that aims to maximize the benefit from single large coalesced memory reads. It will also minimize the amount of local ghost cells required, and thus minimize data transfers. This scheme does however require transposing of the domain between the calculations of F- and G-fluxes. It also has more launches of the time integration kernels; where the original implementation has 2 per timestep this implementation has 4. This implementation will be faster only if the time saved on minimized data reads and simpler flux calculation kernels outweigh the extra time integration steps and the transposition.

The original schemes have an early-exit version of its kernels, as described in 2.2. Neither of the new schemes have any sort early-exit. This is because the early exit in the original scheme was based on a 'wetness map' updated by the time integration kernels. This 'wetness map' consists of one point per thread-block, and so will not work for schemes where the blocksizes are different for the flux-calculation and time integration kernels, as that would cause a mismatch between which cells that are considered wet by the two different kernels.

3.2 Verification

This section will feature verification of the new schemes through centreline-plots. We will present mixed- and second-order separately.

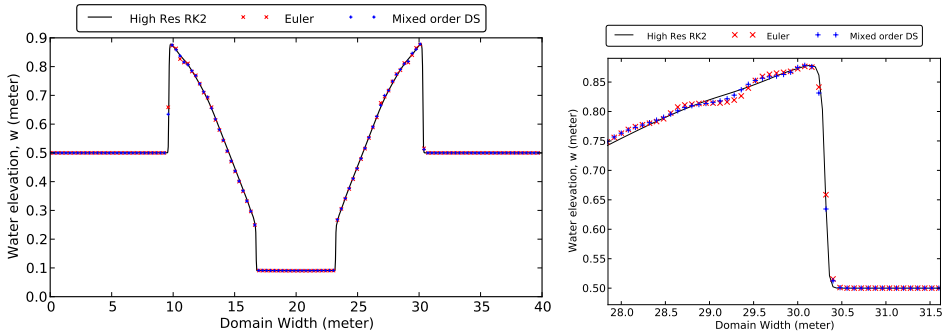
The case run for this verification is a variation on the circular dam break

described in 3.1.2. In the centre of a domain with size $40m$ x $40m$ there is a cylinder of water, with $R = 2.5m$, which is centred at $x_c = 20m, y_c = 20m$. Initial conditions are

$$h(x,y,0) = \begin{cases} h_{ins} = 2.5m & if (x-x_c)^2 + (y-y_c)^2 \leq R^2 \\ h_{out} = 0.5m & if (x-x_c)^2 + (y-y_c)^2 > R^2 \end{cases}, \tag{3.5}$$

and $u = v = 0$ for the entire domain. All calculations done with a resolution of 500^2 and compared to the original Runge-Kutta 2 scheme with a resolution of 1000^2, labelled 'High res'. Boundary conditions are set to wall, which was further described in section 2.3.

Figure 3.4 shows two plots of w, from the case described above, for the two mixed order schemes, Euler and dimensional split mixed order. The leftmost plot is an overview, while the right is a zoomed-in view featuring the rightmost wave front. The wave front plots clearly show the difference in accuracy between the mixed order schemes and the reference second order scheme. The two mixed order schemes are close to identical, and with the Euler scheme having been verified in [2], this supports the accuracy of the mixed order dimensional split scheme.



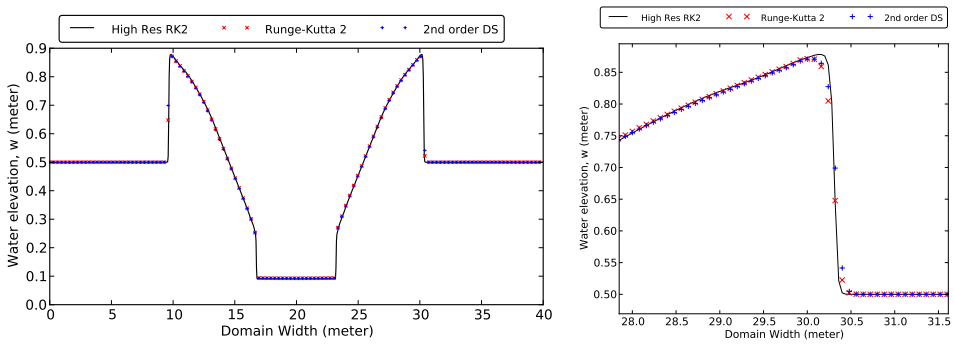
(a) Overview (b) Right wave front

Figure 3.4: Centreline plot of mixed order accurate simulation of idealized circular dam break, which features a cylinder of water with $R = 2.5m$ in the centre of a domain of $40m$ x $40m$. The Euler and DS schemes are run with a resolution of 500 x 500, while the RK2 'High res' is run at 1000 x 1000. The Euler and dimensional split scheme are very similar in accuracy.

Figure 3.5 shows the same plots as described above for the second order schemes. Here the dimensional split and Runge-Kutta 2 schemes are equally accurate. The Runge-Kutta 2 scheme was verified second order accurate in [2]. The scheme with separate flux-calculation kernels was omitted since it numerically is the same as the Runge-Kutta 2 scheme.

3.3 Performance

In this section the performance of our alternate schemes will be presented, and compared to the original schemes. An important factor of the performance is the blocksizes. The blocksizes chosen for the kernels can be



(a) Overview (b) Right wave front

Figure 3.5: Centreline plot of 2nd order accurate simulation of idealized circular dam break, which features a cylinder of water with $R = 2.5m$ in the centre of a domain of $40m$ x $40m$. The Runge-Kutta 2 and 2nd order DS schemes are run with a resolution of 500 x 500, while the RK2 'High res' is run at 1000 x 1000. The two schemes are similar in accuracy with the RK2 scheme having been verified second order in [2].

seen in table 3.1. The ones for the existing schemes' kernels are the same as set by Brodtkorb et al. in [2]. The ones for the newly implemented schemes were set based on minor manual tuning on the Q1800M.

Blocksizes	FGH	RK2	Fx	Fy	Y_block
Height	16	16	64	64	16
Width	12	12	1	1	16

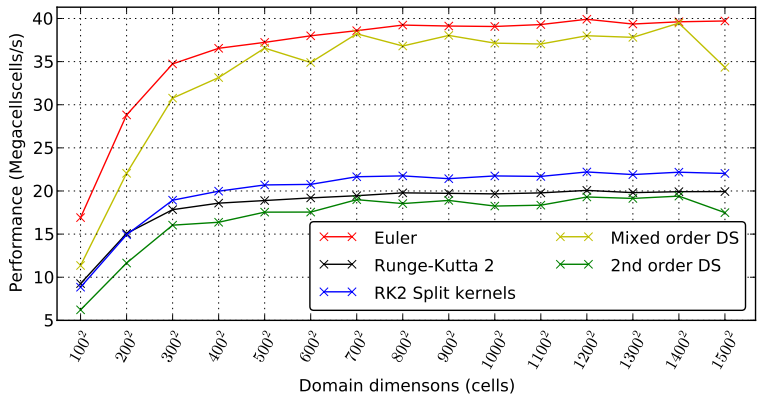
Table 3.1: The reference blocksizes. FGH and RK2 blocksizes are the ones found by manual tuning by Brodtkorb et al.[2]. Blocksizes for the kernels implemented in this thesis were chosen based on manual tuning by the author.

Figure 3.6 shows performance for all schemes on the Nvida Q1800M and GTX480. The case used is the circular dam described in section 3.1.2, cdam(x, t=1), where x is varied and shown along the X-axis of the figures. We can see an increase in cells per second for resolutions up to approximately 500^2 for the Q1800M, and approximately 1000^2 for the GTX480. This increase is due to the GPU's resources not being fully saturated, the domain is not large enough to occupy all the processors and their pipelines. This is the reason GPUs are best utilized for large problems.

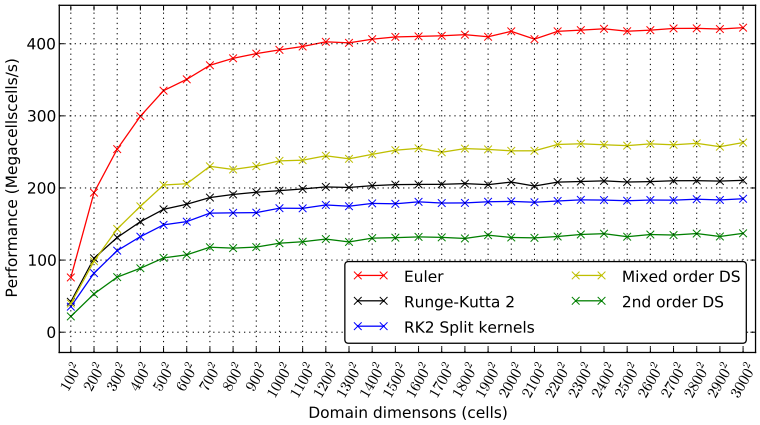
We can see the mixed order dimensional split scheme's (Mixed order DS) expected inefficiency on the GTX480 due to the high rate of transpositions. On the Q1800M the performance loss is much smaller, likely because the weaker laptop GPU's limiting factor is the computations, not the memory transfers. The second order dimensional split (2nd order DS) does also rival the performance of the reference Runge-Kutta 2 (RK2) scheme on the Q1800M. The figures does not capture the fact that the dimensional split schemes calculates more timesteps, because of the scaling

for approximation of Δt, they only show the performance per timestep. Complete performance as a function of execution time will be presented later in this thesis, after auto-tuning have been applied to all schemes.

In the same figure we can also see that the split kernel Runge-Kutta 2 (SRK2) scheme is performing better than the RK2 scheme with the manually selected blocksizes on the Q1800M. Whether near optimal blocksizes were selected, or the scheme in it self is better performing for this GPU will be evident once all kernels and schemes have been auto-tuned.



(a) NVIDIA Q1800M



(b) NVIDIA GTX480

Figure 3.6: Performance of alternate schemes before tuning, simulating a circular dam break on flat bathymetry. The GTX480 has more data points as it is has more memory and thus can run larger simulations. Performance increases with domain size until the GPUs are fully utilized. On the Q1800M we see that the split kernel scheme is outperforming the other schemes by about 20%. For the GTX480, the original schemes are the best performing.

Chapter 4

Auto-tuning

As explained in the introduction, auto-tuning is important to achieve high efficiency for programs intended for a variety of NVIDIA GPUs. This chapter will describe how, and how well, our auto-tuning system addresses this need.

After the implementation of the alternate numerical schemes we have 20 kernels we wish to tune to get as high performance as possible. These 20 kernels are variations of the flux calculation and time integration kernels, described in Section 2.2 and Chapter 3. Each of these 20 kernels has a search space with between 24 and 2048 combinations. To auto-tune these we chose to use empirical search with pruning and micro-benchmarking.

The other part of our auto-tuning is to find the best performing of 3 schemes. We select the best one through benchmarking.

We have limited out auto-tuning to the flux and time integration kernels, as these are by far the most time consuming. The remaining kernels are only a single digit percentage of the total runtime. We also limit our auto-tuning to only the fully second order schemes as those are the most time consuming. This leaves 3 schemes, the Runge-Kutta 2 (RK2), the Runge-Kutta 2 with split kernels (SRK2) and the dimensional split 2 (DS2).

4.1 Exploring the search space

Before applying any sort of auto-tuning we did an exploration of the search spaces to find opportunities for pruning, and other relevant information. For this exploration of the search space we used a python script to repeatedly compile and run our program, and collect the execution times. One kernel's blocksize was altered each time, so that the differences in execution times represent this one kernel only. All figures seen in this section were made from the above mentioned execution times, and are the basis of our pruning of the search space.

The figures of kernels with two dimensional blocksizes have had their runtimes, t', normalized using $t' = t/min(t_1, t_2, ..., t_n)$, so that it is easy to read how much more time non-optimal kernels use as a factor of the most optimal. For the one dimensional kernels we found it more intuitive to

have use execution times as a measure of performance and show it as a percentage of the best kernel.

4.1.1 Flux calculation kernels

Our flux calculation kernels could be categorized as very large kernels with heavy shared memory usage. They are divided into 4 types, as was described in sections 2.2 and 3. FGH which calculates fluxes in both X and Y direction, Fx which calculates fluxes in X direction, Y-block which calculates fluxes in Y direction, and lastly the Fy kernel which calculates fluxes in Y direction on a transposed domain. All of these have two versions for whether they should compute eigenvalues or not (step 0 and 1 of Runge-Kutta 2 time integration).

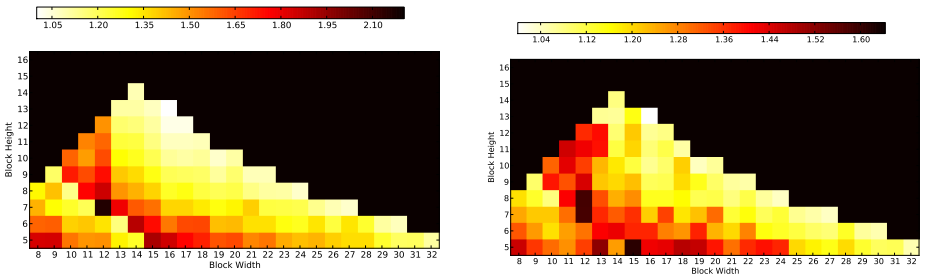
FGH-kernels Figure 4.1 shows execution times with variations in FGH kernel blocksizes.Subfigure 4.1a shows data for variations in blocksize for the kernel with shared memory reduction (step 0), while Subfigure 4.1b shows data for the one without shared memory reduction (step 1). From the colour-bar in the figures we can see that the kernel with no shared memory reduction is less affected by blocksizes, the worst kernel taking 1.6 times longer than the best versus 2.1 for the step 1 kernel. Still they both show the expected pattern of having the best configuration on the boundaries of invalid configurations.

In Figure 4.2 we can see that the GTX480 step 0 and step 1 kernels are much more similar in their pattern than that of the Q1800M, in fact they are nearly identical. The pattern is also more even than for the Q1800M, suggesting that blocksizes make less impact on performance.

We did not do scans of the early-exit versions of the FGH kernels as the only difference is 4 global reads broadcasted to all threads in a block, and a conditional exit based on these values. This is unlikely to change the pattern of the search space in any significant degree.

The search space of the FGH kernels has some 'natural' limitations. They are limited mostly by hard constraints, such as assumptions in their programming that disallows $height > width$ and $n\_threads < 64$. In addition we have the very high shared memory requirements. The FGH kernels end up having 123 valid sizes in total for the Q1800M. Based on the figures we however reduced the search space of the flux kernels further to only blocksizes of $width >= 14$ and $height >= 6$. This was done as the most lenient of the GPUs, the GTX480s, had no near-optimums outside this area.

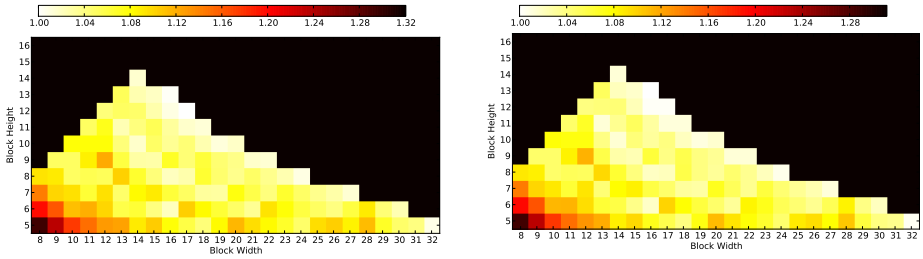
Line kernels Our line kernels, Fx, Fy, both with variations for step 0 and 1, do the same work as the FGH kernels, except they only calculate fluxes in one direction. As they only calculate fluxes in one direction, they only do half the work of the larger FGH kernel. All of these kernels are combined into a single figure per GPU for easy comparison.



(a) FGH kernel with shared memory reduction of eigenvalues.

(b) FGH kernel with no shared memory reduction

Figure 4.1: Blocksize plots of non-early exit FGH kernels on a NVIDIA Q1800M. The best configurations are found at the border of invalid configurations. For both kernels the most optimal configuration is 16x13.



(a) FGH kernel with shared memory reduction of eigenvalues.

(b) FGH kernel with no shared memory reduction

Figure 4.2: Blocksize plots of FGH kernels on a NVIDIA GTX 480. Here too the best configurations are on the border of invalid configurations. For kernel (a) the best configuration is 17x13, while for kernel (b) it is 16x12.

Figure 4.3 shows these kernels for the Q1800M. As we can see all the 4 kernels have much the same pattern. The blocksizes are limited to minimum width of 64, as the shared memory reduction of step 0 kernels assume at least this many threads. Smaller blocksizes are anyway unlikely to yield good performance as there are too few half-warps within each block for good pipeline usage. We can also see that in general the kernels have the same performance for all lengths with the same number of half-warps. The exceptions from this are when the size of the kernel does not well fit the domain size, so that extra blocks have to be launched to calculate only a few cells. This is the explanation for the sudden drops in the middle of half-warp intervals. In these cases the overall performance of each run depends more on a fitting length, which 'wastes' few threads on computations in the padding, than the actual performance of the kernel. This is prominent in said figure since it was run on a small domain. Data was collected by changing the dimensions of only one kernel at a time, but as stated earlier they are displayed in the same plot for easy comparison. We can otherwise see a trend of reduced performance for each added half-warp after the optimal length of 128.

Figure 4.4 shows the same kernels for the GTX480. Comparing them against the same kernels for the Q1800M presented in Figure 4.3, we can see that also on this GPU the performance is generally the same for all sizes with the same number of half-warps. As with the Q1800M, we can see some deviations from the general trend. Some of these exceptions are due to the same cause. This effect is however reduced for the GTX480 as the data was generated by runs on a larger grid to saturate this more powerful GPU. We have a different reason for the deviation from the general pattern on this GPU. This deviation is likely from the addition of a new long size read, which the older Q1800M is not capable of. The difference from the Q1800M is that the four different kernels are not as similar in performance, and that the performance varies less with the blocksize.

Based on these plots, the only pruning we can do of the single height kernels is to limit the search space of these kernels to only full half-warps, i.e. multiples of 16. This reduces the search space by a factor of 16.

Figure 4.5 shows the effect of changes in the Y-block kernels on the Q1800M. Through the middle of this figure we can se a 'belt' of slow kernel sizes. This belt is presumably of kernels of a size where only one block fits per streaming multiprocessor (SM), but is too small to the processor, leaving a lot of resources unused. Unlike with the FGH kernels, here we have the most optimal sizes where 2 kernels barely fit per SM. In general kernels of $width = 16$ performs well.

Figure 4.6 shows the effect of changes in the Y-block kernels on the GTX480. Looking at the row with blocks of height 7 we can clearly see that performance increases with increased width up to a point when it suddenly drops. This drop is likely because the increase in blocksize caused one less block to fit per SM. The pattern repeats, but the performance drop gets less each time. The best performance is still found with blocks of $width = 32$. As a curiosity we also note the slow kernels along the border of invalid configurations.

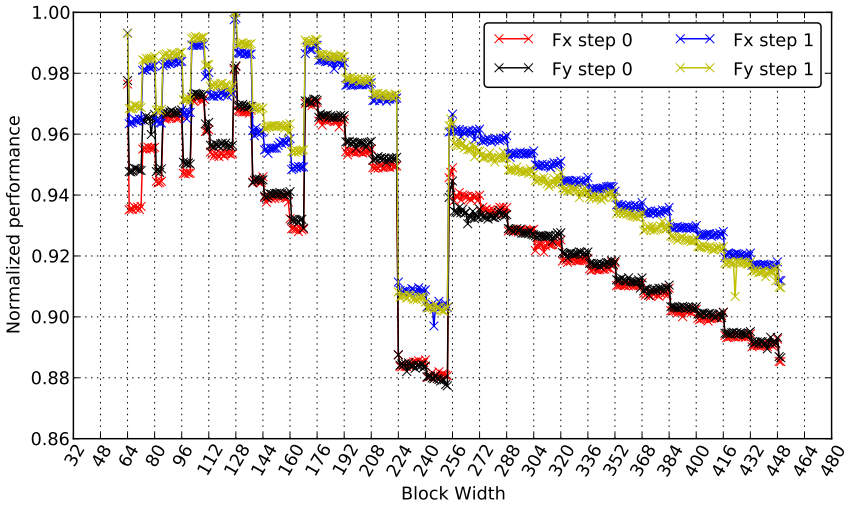


Figure 4.3: Blocksize plots of flux calculation line kernels on a Q1800M. All sizes with the same number of half-warps have roughly the same performance, with the exception of where one additional thread makes it possible to launch one less kernel to cover the domain. Performance is normalized to that of the fastest kernel and size. Best configuration is 128 for all kernels.

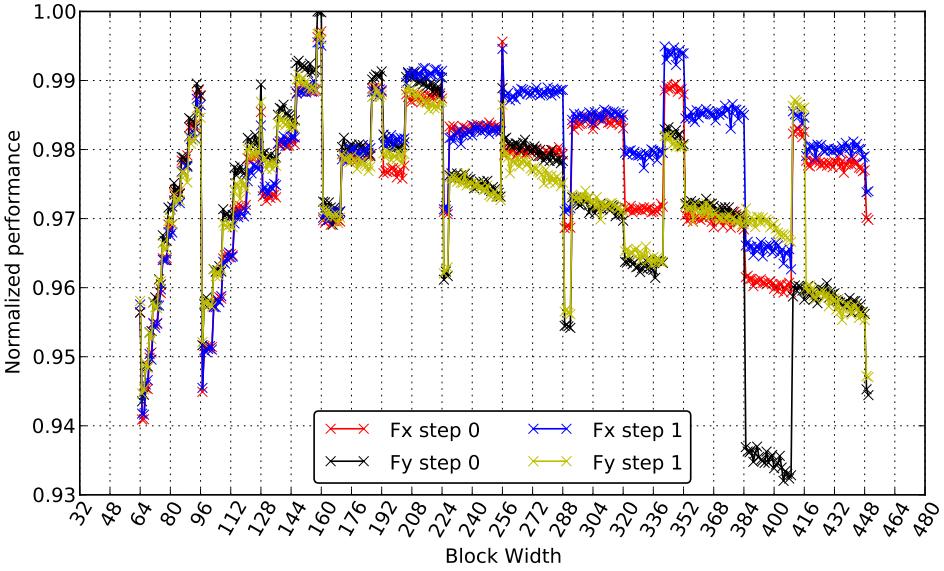
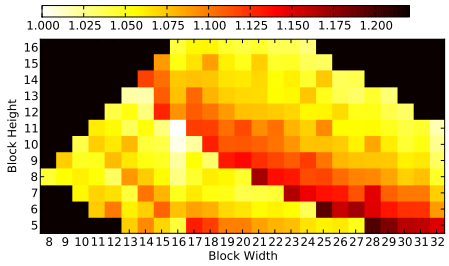
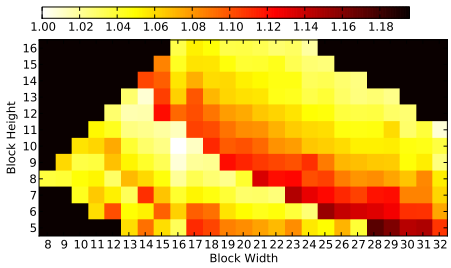


Figure 4.4: Blocksize scan of Fx and Fy step 0 and 1 kernels on a GTX480. Performance is normalized to that of the fastest kernel and size. The best configurations are: 160 for Fx step 0, 159 for Fx step 1, 158 for Fy step 0 and 159 for Fy step 1.

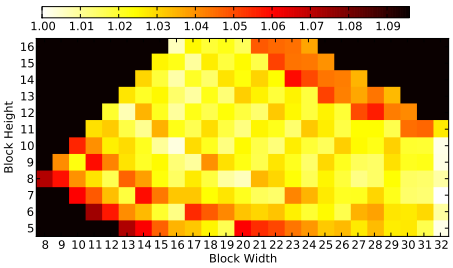


(a) Y-block kernel with shared memory reduction of eigenvalues.

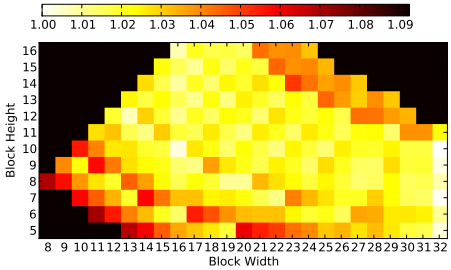


(b) Y-block kernel without shared memory reduction of eigenvalues.

Figure 4.5: Blocksize plots of Y-block kernels on a NVIDIA GTX Q1800M. Y-block kernel with shared memory reduction of eigenvalues. The figure shows a 'belt' of slow kernels. The optimum blocksize is at the border of this belt. The best blocksizes are 16x11 for (a) and 16x10 for (b).



(a) Y-block kernel with shared memory reduction of eigenvalues.



(b) Y-block kernel without shared memory reduction of eigenvalues.

Figure 4.6: Blocksize plots of FGH kernels on a NVIDIA GTX 480. We can see performance increases with added threads, until it take a heavy drop. This is most evident to the lower left in the figures. The best performing kernels are 32x7 for (a) and 32x10 for (b).

As with the line kernels we will prune the search space to only include blocks where the thread count is a multiple of 16, vastly reducing our search space.

From the figures we can clearly see that the flux calculation kernels are greatly affected by the blocksizes. We also note that the flux kernels' performance can be more than halved between the best and worst configuration. Since the flux calculation kernels stand for approximately 80% of total runtime, the change in performance greatly affects overall performance. Tuning of the blocksizes is therefore greatly desired.

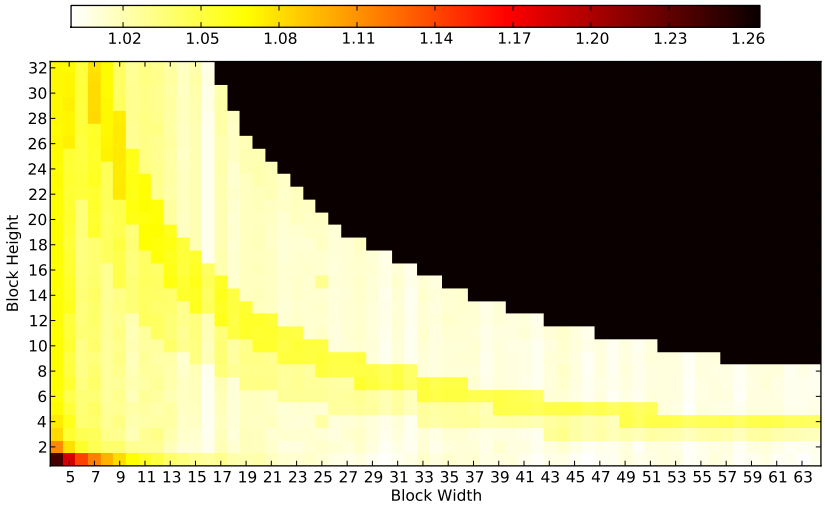
4.1.2 Time integration kernels

Our time integration kernels are fairly lightweight. They do few calculations and 6-15 global reads and/or writes depending on which variation of the kernel it is. None of these kernels, except for the ones updating the wet-map, use any shared memory. Those that do update the wet-map only use one 4 byte (one float) of shared memory per thread. Such low resource usage allows for very large kernel sizes, and therefore yields a very large search space. This also makes the search spaces of the time integration kernels our largest. We have limited our search space scan of these kernels to 64 in width and 32 in height. Tall kernels are unlikely to yield good performance as the NVIDIA architecture is focused on and optimized for long X-direction memory reads. We started our scans at 1x1 threads, but do not display widths smaller than 4 since they are so badly performing that the difference between other sizes get hard to discern.

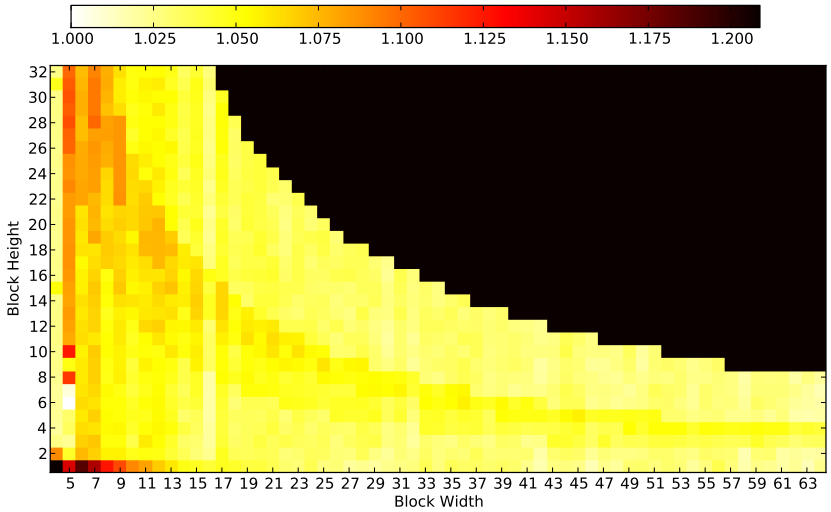
Figure 4.7 shows the blocksize scan for step 0 and 1 of the non-early exit time integration kernels. The only differences between the two kernels in Figure 4.7 is that the bottom one has an additional 8 floating point operations and 3 global memory reads. This however clearly alters the optimum blocksize. The step 0 kernel has its optimum at $width = 63$ and $height = 1$. Even longer widths may be better but where not explored. The step 1 kernel has its optimum at $width = 5$ and $height = 6$. Both kernels have a near-optimum at $width = 16$ and $2 < height < 8$. This near optimum is approximately a 2% performance loss for the step 0 kernels, and approximately a 1% loss for the step 1 kernel, compared to the optimal blocksize. These losses are for the entire scheme when changing only the blocksize of the related RK kernel.

Figure 4.8 shows the same plots for the early-exit kernels. These kernels are the same above, except they also do a shared memory reduction as a part of updating the wet-map for early-exit. In the figure it looks like the pattern is greatly changed, but this is merely due to the colour bar changing as slow blocksizes are eliminated by the requirement of minimum 64 threads for the shared memory reduction. The worst blocksizes in this figure is not more than 10% slower than the optimal blocksize.

Figure 4.9 shows the Runge-Kutta 2 time-integration kernels without wet-map updating for a GTX480. We can see that blocksizes with few threads greatly reduces performance. For blocksizes with more threads the performance varies little.

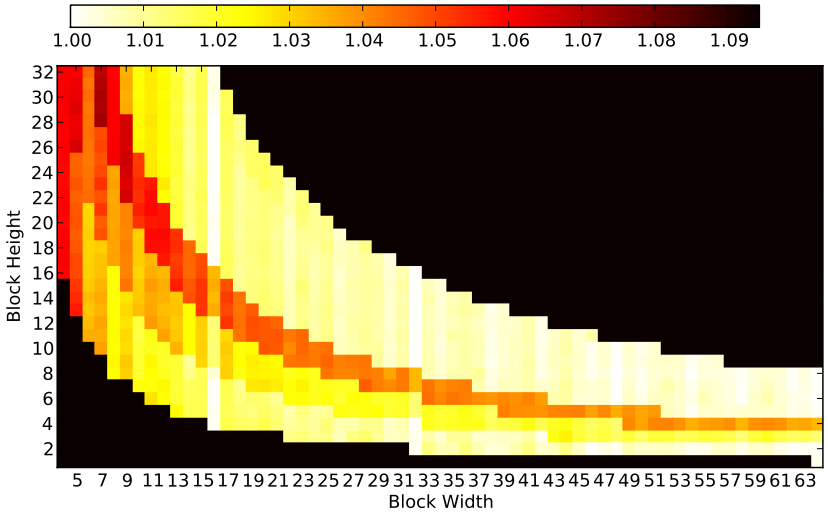


(a) RK kernel for Euler and first step of RK (step 0 kernel).

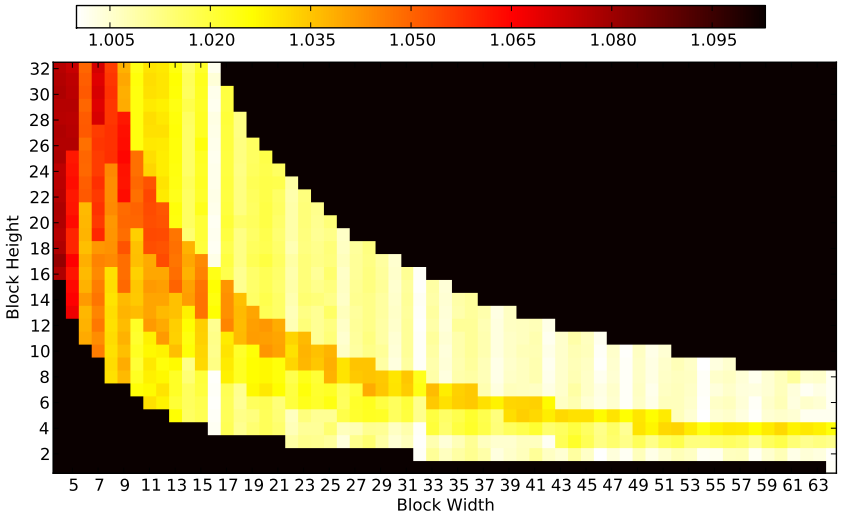


(b) RK kernel for second step of RK (step 1 kernel).

Figure 4.7: Blocksize plots of RK kernels without the wet map shared memory reduction (non-early exit) on a NVIDIA Q1800M. The performance for these kernels is very even. Almost all blocksizes are within 8% performance loss compared to the best configurations of 63x1 (a) and 5x6 (b).

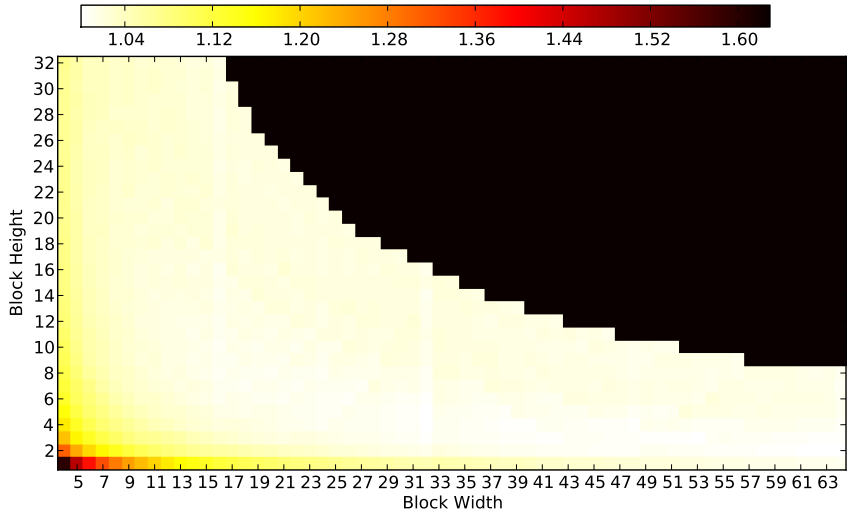


(a) RK kernel for Euler and first step of RK (step 0 kernel).

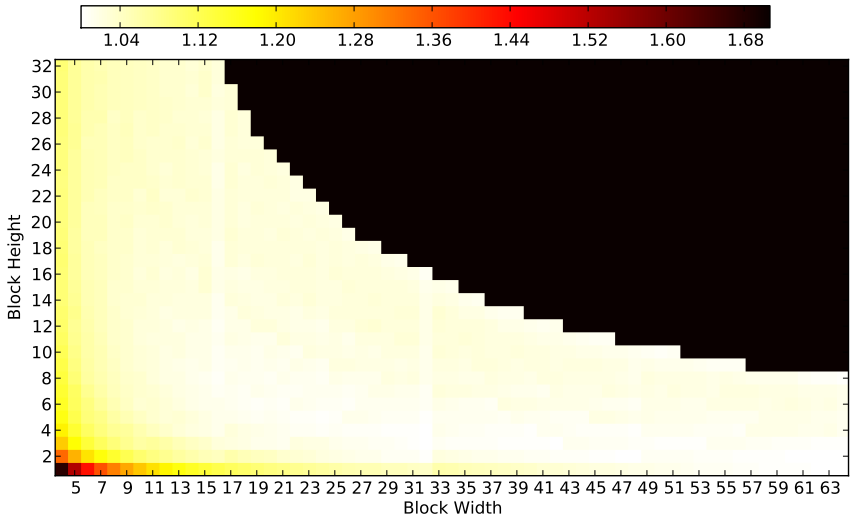


(b) RK kernel for second step of RK (step 1 kernel).

Figure 4.8: Blocksize plots of RK kernels with the wet map shared memory reduction (early exit) on a NVIDIA Q1800M. Here too the performance is very even. The worst performance is only 10% slower than best configurations of 32x14 (a) and 46x2 (b).



(a) RK kernel for Euler and first step of RK2 (step 0 kernel).



(b) RK kernel for second step of RK2 (step 1 kernel).

Figure 4.9: Blocksize plots of RK kernels *without* wet map shared memory reduction on a NVIDIA GTX480. Here we see that as long as enough threads are present, the blocksize has little impact. The best blocksizes are 64x2 and 62x2 for Sub-figure 4.9a and Sub-figure 4.9a respectively.

For the same plots of the wet-map updating kernels, Figure 4.10, we can see a rather different pattern. Instead of the 'belt' half way between the smallest and largest kernels, we here have a more chequered pattern. We can see the performance is best for sizes where the number of threads is a multiple of 32. This is particularly clear if you follow the row of $height = 2$. Since the height is 2 each increase in width adds 2 threads, hence the multiple of 32, not 16.

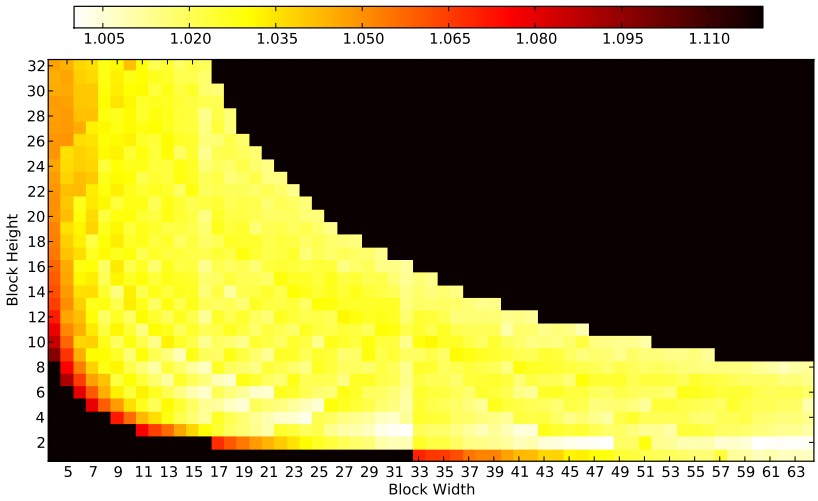
Figure 4.11 and 4.12 shows blocksize plots for the 4 transposition kernels used in the dimensional split scheme for the Q1800M and GTX480 respectively. These kernels vary only with one variable since transposition requires a perfect square of shared memory. In our case the shared memory is linked to the number of threads so that we also get a square number of threads. It is interesting to note that for the older Q1800M the optimal is 18, while for the GTX480 it is 16. This may be because of changes in features of the shared memory's banks, which results in less bank conflicts for the newer GTX480. In these figures we can also see that these kernels do not vary greatly in performance for blocksizes of 16 or larger. We simply set their blocksizes to 18 for compute capability 1.X cards, and 16 for any others.

We apply this limitation the micro-benchmark search space as well. Long kernels would run into the same domain-fitting problem as the Fx and Fy kernels. These kernels use no, or little, shared memory, and so the GPU is free to do long reads and utilize the L1 cache. This should reduce or remove the performance benefit long kernels otherwise might have had. As with the line kernels it is plausible to limit the search space to only full half-warps, greatly reducing our largest search space. From the figures we can also see that the time integration kernels are less affected by the blocksizes. For them the difference in performance is at worst 'only' 25%. If the search space is limited to $width >= 14$ and any $height$, the difference between the best and worst kernel is only around 10%. Since the time integration kernels makes up approximately 15% of total kernel time, the impact from tuning theses kernels is rather low (1.5%).

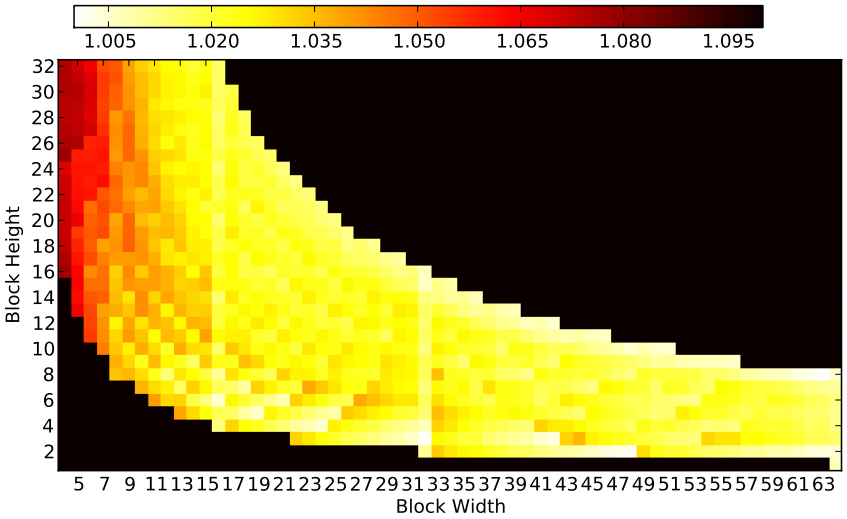
4.2 Micro-benchmarking kernels

As was stated in the introduction, our problem is twofold. One part of the problem is the selection of blocksizes, which will be addressed in this section. To find the optimal blocksizes we explored the possibility of using empirical search with micro-benchmarking. We expected this to be feasible since the optimal blocksize for each kernel is not dependent on the input dimensions, and therefore one sweep of the blocksize space is enough. We could also greatly alleviate this problem by timing several kernel types each timestep. Each timestep of a single scheme, we are able to time one blocksize for each of the flux and time integration kernels, of that scheme. If we do this for the Runge-Kutta 2 scheme, the split kernel scheme and the dimensional split 2 scheme, we would cover all our kernels.

Doing benchmarking of the kernels by redefining cmake defines of



(a) RK kernel for Euler and first step of RK2 (step 0 kernel).



(b) RK kernel for second step of RK2 (step 1 kernel).

Figure 4.10: Blocksize plots of RK kernels with wet map shared memory reduction on a NVIDIA GTX480. The general trend of the data is that performance gradually increases up to a full half-warp and then drops heavily once a new half-warp is required.

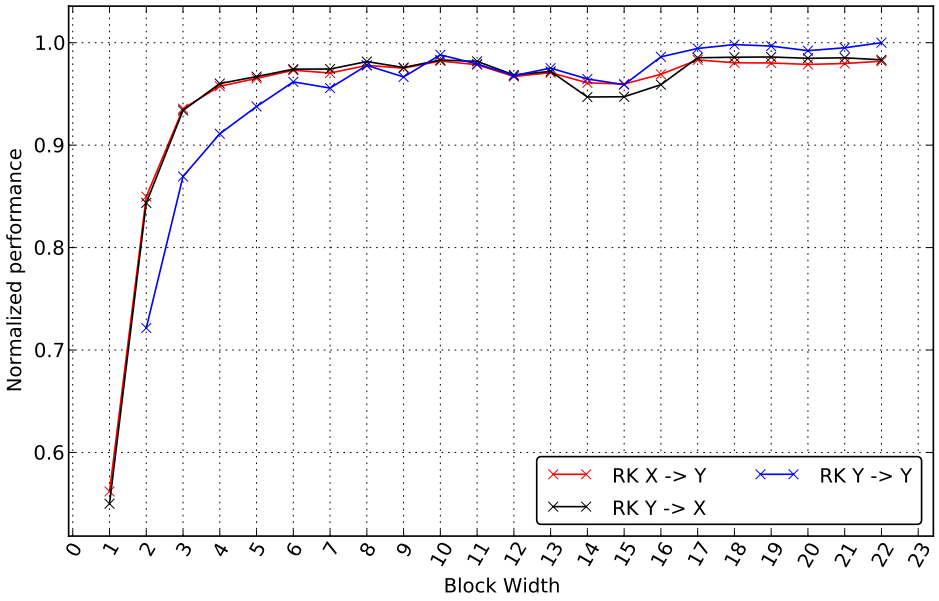


Figure 4.11: Blocksize plots of RK transposition kernels on a NVIDIA Q1800M. The performance varies little for dimensions of 17 or larger.

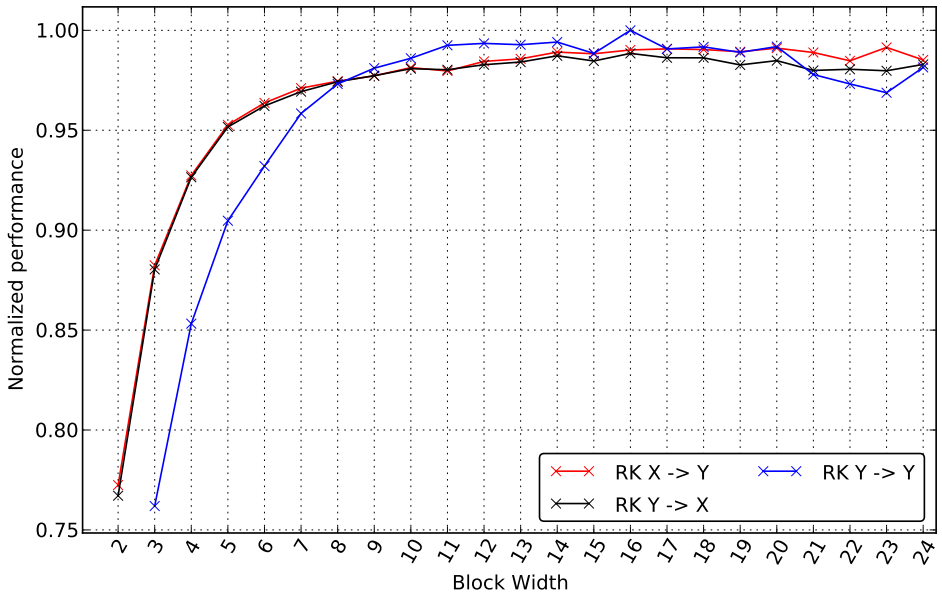


Figure 4.12: Blocksize plots of RK transposition kernels on a NVIDIA GTX480. The performance is with 3% of the best for all dimensions of 8 or larger.

kernels sizes and recompiling the implementation would take a very long time. This is what was done for the blocksize plots in Section 4.1. For the Runge-Kutta kernels (the largest search space) such an approach took about 6 hours each. If this was done for all the kernels it would take days, and quite clearly not be a feasible way to auto-tune our program, as such time-scales would not be worth the performance gained. To avoid this problem we used the cuda driver API and implemented a runtime compilation of our kernels along with logic that make sure we do it in as few timesteps as possible. We found runtime compilation of a kernel to take about one second, versus the 30-40 second to compile the entire program. We also save considerable time applying the pruning of the search space found possible through the search space exploration in the previous section.

Another time-saving element is our use of micro-benchmarking. Instead of timing entire simulations we timed singular kernel launches. The framework we implemented to do this also allowed us to time all the kernels in a scheme in a single timestep.

4.2.1 Compiling and loading ptx

Runtime compilation of kernels has to be done via the ptx format. Ptx is an intermediate pseudo-assembly format nvidia uses. C/C++ kernel-code is compiled into ptx. This ptx is then compiled into actual assembly. Ptx is needed as an intermediate format as the different architectures have different hardware instructions.

Sadly this ptx format has no support for variables/macros or the like. It is therefore unable to compile C++ templates(without defining each parameter), or act as a template itself. To overcome this obstacle we converted our C++ kernel-templates into regular functions with blocksizes and other parameters defined by macros. This way we can compile them using *nvcc* at runtime and specify the macros through command line defines[1]. By having them macro based we also avoid the possible problem of function name mangling that C++ compiler applies to compiled template functions. This could have been a problem as we need to name the functions to load them at a later stage.

The compilation from source to ptx using *nvcc* takes 1 second for one of our flux kernels. This adds some overhead, but is significantly faster than recompiling the entire implementation. The ptx is then JIT compiled, loaded and executed with the desired parameters through the use of the cuda driver API. The overhead for JIT compilation and loading of the .ptx file is on the order of 10^{-1}seconds, which is negligible.

4.2.2 Collecting metrics

The runtime compilation of kernels described above allows us to go through the entire pruned search space of possible blocksizes in a single

[1]Example: nvcc -ptx /path/kernel.cu -Dthreads_x=20 -Dthreads_y=10 -o /output-path/kernel.ptx -L "lib1,lib2,..." -arch="sm_..."

run of the simulator. A sub-class of the simulator is used for benchmarking the kernels. This sub-class works as described in the pseudo-code below:

```
while (more kernels) {
  getNextBlocksizes();
  if (changed sizes in RK2) {
    for (i=0 ; i<10 ; i++)
      RK2Step();
  }
  if (changed sizes in SRK2) {
    for (i=0 ; i<10 ; i++)
      SRK2Step();
  }
  if (changed sizes in DS2) {
    for (i=0 ; i<5 ; i++)
      DS2Step();
  }
}
```

The DS2 step is only run half the number of times of the other schemes as it does two timesteps each time. By running the Runge-Kutta 2, Split kernel and dimensional split 2 schemes we run all the possible kernels.

To handle the selection of blocksizes we created three C++ classes. The one interesting here is BlocksizeLearner. Each time the learner (getNextBlocksizes() in the code above) is called it sets the blocksizes for each scheme to the next blocksize to be benchmarked. It takes care to not miss any blocksizes in the case where a scheme hits an invalid kernel, as invalid kernels exits the step-iteration loop for that scheme.

This class also times the kernel executions using the CUPTI callback API. By using CUPTI we get the timestamps as close to the start and end of kernel execution as possible so that as much interference as possible from process context switching and such is eliminated. It is still possible to get serious interference, see section 4.2.4 for more on this issue and how it was solved.

To get accurate and stable results it is also important to do this benchmarking simulation on a domain that is big enough to saturate the resources of the GPU. The resources of the GPU can be said to be saturated when the graph in Figure 4.13 reaches near maximum cells per second. Before this point the overhead is large compared to the actual computations. Based on Figure 4.13 we selected $dx = dy = 500$ for the Q1800M. Using a larger size would only take more time than necessary.

To further increase accuracy we first run the benchmarking binary (*kp_learner*) once to compile all the kernels. This takes a little more than 20 minutes. We then discard the data on execution time of the kernels as these can be unstable due to the source-to-ptx compilation. Lastly *kp_learner* is run 5 times so that we get 50 timings of each kernel total. Spreading the timings over more than one run should provide additional robustness against interruptions and background operations by the operating system. A full scan of all blocksizes with compilation to ptx takes approximately 15

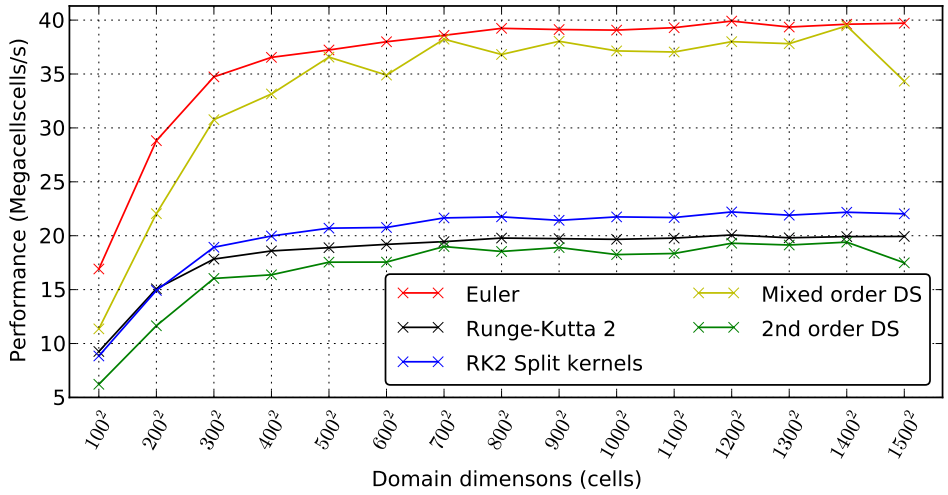


Figure 4.13: Cells per second for a Q1800M on various blocksizes. We can see that the graph levels out at dimensions of proximately 500 (0.25 mega cells total) and out. We can also see the performance of the three schemes prior to tuning, with the RK2 split kernels scheme performing the best on this GPU.

minutes. When the ptx-files exist, a scan takes roughly 7 minutes.

4.2.3 Using the collected metrics

The collected data, which is 50 execution times for each kernel and blocksize, is handled in a python script. This script calculates the standard deviation of each kernel/size based on the 50 values. Any value outside 3 standard deviations is discarded as anomalies, and the mean calculated over the remaining values. Based on these means the best blocksize for each kernel is selected and written to a text file for use by the BlocksizePredictor. Upon launch of the simulator, this predictor reads the stored sizes and passes them to the simulator so that any kernel run will be with these dimensions.

4.2.4 Unforeseen issues

During our empirical search and micro-benchmarking we encountered two major unforeseen issues that were difficult to overcome. These two issues are sharing the GPU with the operating system (or other programs) and different outcomes of different ptx compiler versions.

Sharing the GPU Linux uses the graphics card for desktop rendering at all times. This creates a possibility that an OS kernel may be running at the same time as a computational kernel. This may degrade performance and thus cause incorrect results when doing micro-benchmarks. It may also cause very unstable runtimes, as the OS kernels may not interfere with

every kernel launch. Both of these issues can be overcome by temporarily turning off Linux' desktop manager and running the micro-benchmarking from a terminal-only setup[2], or by using a different graphics processor for the desktop manager[3]. Running the kernel micro-benchmarking with the desktop manager on, yields results where the kernel execution times often are twice that of the shortest execution. With a terminal only micro-benchmarking the values are remarkably stable. Standard variation is at its worst only 10% of its respective averages for 50 executions of each kernel, and 98% of the total values are within 3 standard variations.

Ptx compilation target The NVIDIA ptx compiler, nvcc, produces different ptx-code depending on which GPU is targeted. Kernels' code seems to be optimized much better by the compiler when the intermediate ptx is compiled for sm_10 as compared to sm_20 for the GTX480. Compiling for a different target will cause a different version For example, using sm_10 on the GTX480 halved the computation time of the SRK2 kernels, and caused them to use 10 less registers, even though the GTX480 is a sm_20 GPU. Compilation flags were not explored further, but are noted as a topic for future research.

4.3 Impact of blocksize tuning

In this section we will present data on the speed-up achieved by auto-tuning kernel blocksizes on the Q1800M, the GTX480 and the GTX435M. The figures presented show the ptx version of the three schemes' performance improvement against themselves. We did not compare them against the non-ptx schemes as we have a 0.3 second overhead of loading the ptx files which would make the speed-up unclear and varying depending on the length of the simulations. The speed-ups presented in this section are 'relative speed-ups' as described in [17], as a percentage.

The case used for these performance results is a Gaussian water peak in the centre of a domain. The domain has $length = width = 253.5m$ and is calculated on a resolution of 507^2, and thus $dx = dy = 0.5m$. The water peak's defining function is $w = 3e^{50(-x^2-y^2)}$, where $x \in [-80, 80]$ and $y \in [-80, 80]$, and is centred on the domains centre. Simulation was run until $t = 2$

Figure 4.14 shows the performance of the Runge-Kutta 2 (RK2), split Runge-Kutta 2 (SRK2), and dimensional split 2 (DS2) schemes after blocksize tuning on a NVIDIA Q1800M. This figure does not show each kernels actual performance, but its increase over itself after the blocksize tuning. For the Q1800M we see that the RK2 scheme and the DS2 schemes have close to no significant performance increase. It is likely due to the default blocksizes being near optimum, so there is not much improvement to be found.

[2]For example by CTRL+ALT+F1 on Ubuntu.

[3]Such as the integrated GPU newer CPUs have, e.g. Intel HD Graphics introduced in Jan '10, or another graphics card in multi-GPU setups.

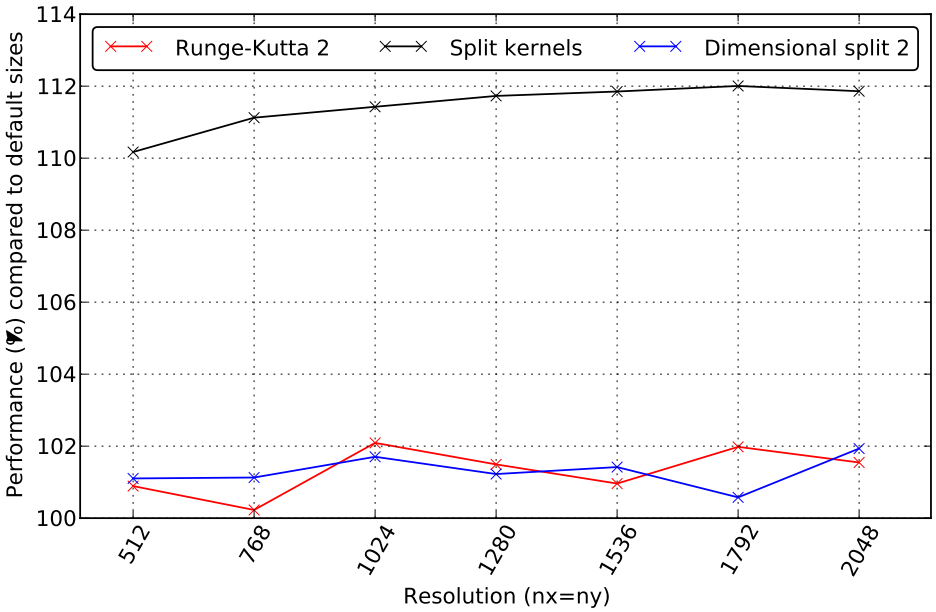


Figure 4.14: The effect of kernel blocksize micro-benchmarking on a NVIDIA Q1800M. Tuning of the RK2 and dimensional split did not have significant impact for this GPU, and suggests that our reference blocksizes are near optimal for this GPU. The SRK2 scheme did however see a significant (10-12%) performance increase.

Speed-up for the GTX480 can be seen in Figure 4.15. For this GPU we can see performance increases of up to 20%, with the RK2 scheme having the smallest increase of close to 10%. The sharp increase in performance of the SRK2 and DS2 schemes between the resolutions of 1024 and 1280 is likely due to the GPU not being fully saturated at 1024.

Figure 4.16 shows the same data for a GTX435M. This GPU was not used in the development and therefore is a test of how our auto-tuning performs on a GPU that was not considered. This GPU can be said to be a mix of the Q1800M and GTX480. It is the same generation (compute 2.1) as the GTX480, but like the Q1800M it is also laptop model with much fewer cores compared to desktop cards such as the GTX480. Across the GPUs this one had the greatest increase in performance ranging from proximately 12% to 33%.

For all three of our GPUs the RK2 scheme has the lowest performance increase. This is not unexpected as the reference blocksizes for this scheme have been manually tuned by Brodtkorb et al. in [2]. The other reference blocksizes was selected by the author based on knowledge of the architectures.

See table 4.1 for an overview of the blocksizes before and after tuning. This table verifies that different graphic card models has different optimum blocksizes, and shows that tuning is necessary. We also note that some of these blocksizes were marked as not valid in the exploration of the search spaces earlier in this chapter, and is due to different compilation

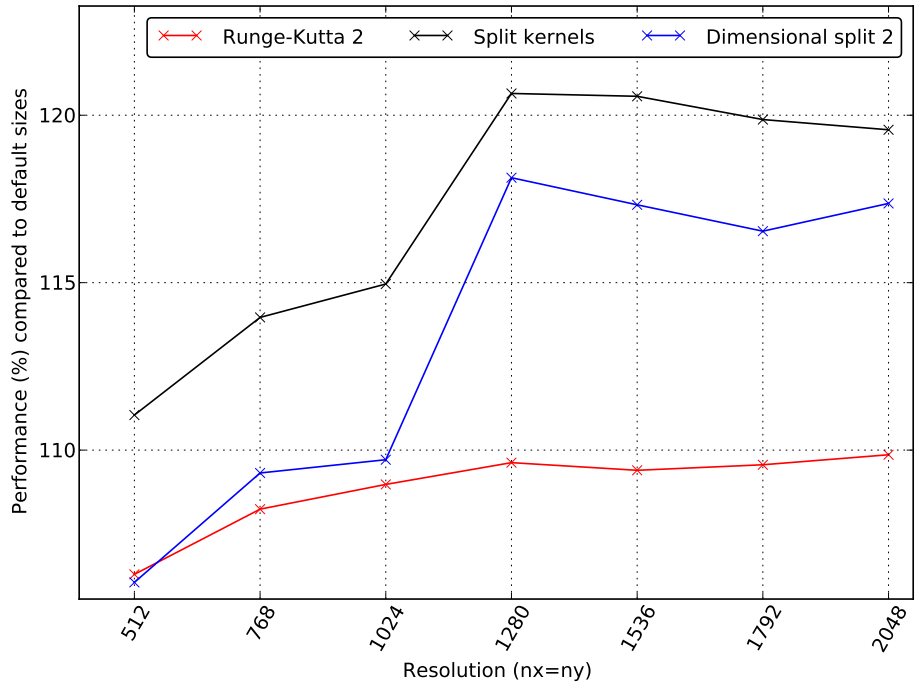


Figure 4.15: The effect of kernel blocksize micro-benchmarking on a NVIDIA GTX480. We can see a significant increase in performance for all schemes.

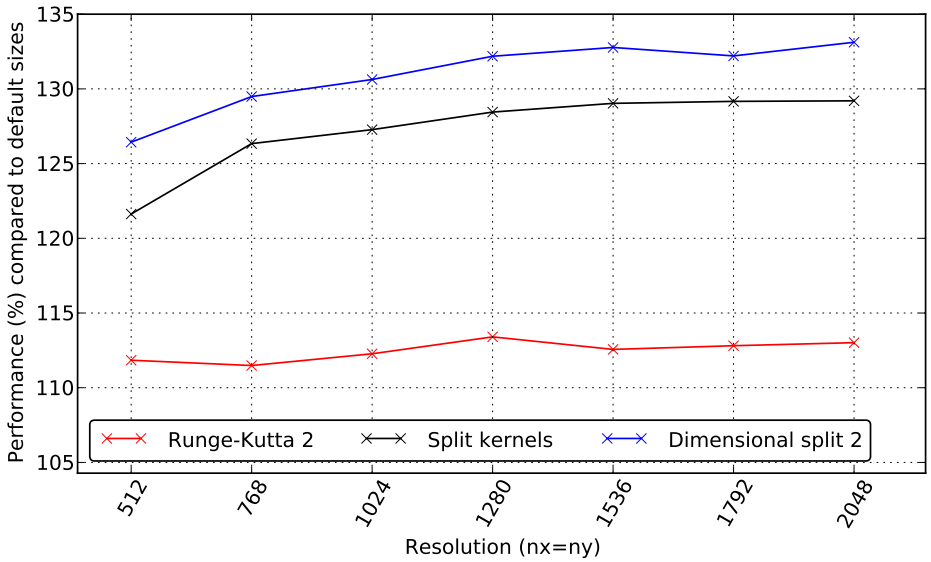


Figure 4.16: The effect of kernel blocksize micro-benchmarking on a NVIDIA GTX435M. There is a significant (12-33%) performance increase for all schemes.

parameters.

Kernel	Reference	Q1800M	GTX480	GTX435M
FGH0	16x12	16x13	32x10	32x10
FGH1	16x12	16x13	32x10	32x10
Y_block0	16x16	16x11	32x7	32x7
Y_block0	16x16	16x11	32x7	32x7
X_line0	64x1	128x1	256x1	256x1
X_line1	64x1	128x1	336x1	256x1
Y_line0	64x1	128x1	144x1	128x1
Y_line1	64x1	128x1	144x1	128x1
RK0	16x12	16x16	64x2	32x6
RK1	16x12	32x14	64x2	64x2
SRK0	16x12	32x8	64x2	64x3
SRK1	16x12	32x8	64x2	64x3
EE0	16x12	16x13	32x10	32x10
EE1	16x12	16x13	32x10	32x10
Performance gain RK2	-	1.5%	9.9%	13.0%
Performance gain SRK2	-	11.9%	19.6%	29.2%
Performance gain DS2	-	1.9%	17.4%	33.1%

Table 4.1: The reference blocksizes, and the blocksizes selected after micro-benchmarking on a Q1800M, GTX480 and GTX435M. The performance gains are from the highest resolution simulation in each of the Figures 4.14, 4.15 and 4.16.

4.4 Auto-tuning system

This section will describe the remainder of our auto-tuning system. Figure 4.17 shows the modules in our auto-tuning system and their dependencies. First the parts of this system will be described separately, and then three use cases will be presented to illustrate how the parts interoperate.

The Modules The system consists of several modules, where *the tuner*, the *Blocksize selector* and the *Kernel handler* are the most interesting with regards to the auto-tuning. The other modules are related specifically to the shallow water simulator.

The *Blocksize selector*'s function is to choose the blocksize each kernel should use. It has three modes, the benchmarking mode, a tuned mode, and an un-tuned mode. In un-tuned mode it uses the user defined blocksizes from *cmake*. In tuned mode it reads the best blocksizes, and the best scheme, found through auto-tuning from file. When running in benchmarking mode, it iterates through the predefined blocksize search space, and uses CUPTI to micro-benchmark kernels as described previously in this chapter. The results of the micro-benchmarking are written to disk, in separate files for each kernel.

The *Kernel handler* is the module which does the actual kernel invocations. When a kernel is requested to be launched, this module checks if the requested kernel is already loaded, in which case it is launched with the blocksize specified in the request. If the kernel is not loaded, the Kernel handler looks for a corresponding ptx file, and loads it from that file. In the case the ptx file does not exist either; the Kernel handler compiles a new ptx file from source using nvcc.

The *Tuner* is the module which ties the system together. This module is written in python and operates by launching applications which uses the simulator, and by reading and writing data to disk. When executed the Tuner runs the Kernel Micro-benchmarker to generate kernel timing data. This timing data is read from disk, the best of (fastest) blocksize for each kernel is found as described in Section 4.2.3, and written to a new file. Next, the Tuner executes the simulation application once for each scheme, with the schemes specified through parameters. Each of these executions is timed, and the fastest selected as the best scheme. The best scheme is written to the same file as the best blocksizes for later simulations.

Use cases Three use-cases will be presented to describe how out auto-tuning system operates. One case will be the tuning of the kernels, the second will be benchmarking of the schemes, and the third will be a normal simulation with early-exit. The first and second cases are run after each other by the Tuner.

Before kernel micro-benchmarking is started, the Kernel Micro-benchmarker should be run once to produce all the required ptx files. The csv-files with kernel timing data, which also are produced by this run, should be removed as they can have unreliable micro-benchmarks. Then the python module 'tuner' should be executed, which in turn will run the Kernel Micro-benchmarker 5 times. The Kernel Micro-benchmarker is a c++ application using a sub-class of the shallow water simulator which gives access to the functions required to benchmark kernels. The shallow water simulator then gets the next kernel blocksizes from the blocksize selector, and executes the kernels using the kernel handlers. The blocksize selector also has CUPTI callbacks enabled which times the kernels. The gathered kernel execution times are written to csv-files which we will call the timing dump. The shallow water simulator continues to get blocksizes from the blocksize selector, and executing kernels with these blocksizes, until the blocksize selector has iterated through all the desired blocksizes. Once the kernel micro-benchmarker has been run five times, the tuner reads the entire timing dumps and finds the best blocksize for each kernel, and writes these to another csv-file. The process so far was described in Section 4.2 in greater detail.

The second use case is the benchmarking of the full schemes. The tuner initiates this once kernel micro-benchmarking is done by running the simulation application with specific schemes, and timing the runs. In this mode the shallow water simulator only gets blocksizes from the blocksize selector once, and uses them for the entire simulation. The blocksize

selector reads the blocksizes from the csv file generated by the tuner. The fastest of the timed schemes is written to the same csv-file as the blocksizes.

The third use case is a normal simulation with early-exit. As with the last use case, the simulator gets the blocksizes once from the blocksize selector, which gets them from the csv file. In this mode the *preferred scheme* is also read from the same file. The shallow water simulator will run both the read *preferred scheme* and the early-exit solution, and time them. For the next 97 timesteps the fastest one of the two will be used, then they both will be timed again etc. until the simulation has reached its end-condition.

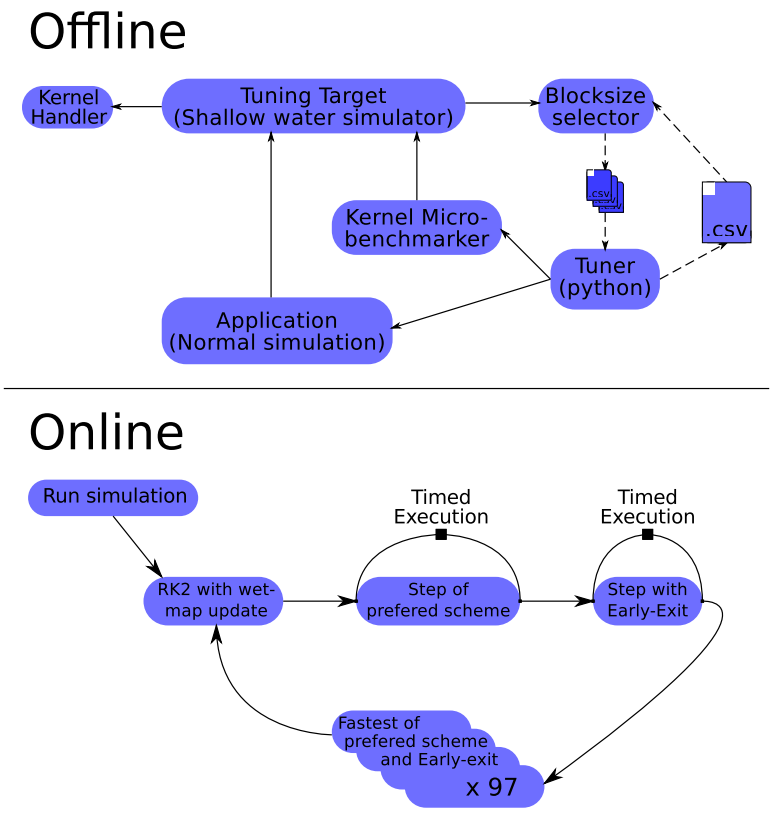


Figure 4.17: The modules of our auto-tuning system, and the dynamic selection of scheme. Solid arrows indicate dependencies, while dashed arrows indicate reading from or writing to file.

4.4.1 Finding the fastest scheme

To find the fastest of the now tuned schemes, we use a benchmarking approach. We also limited ourselves to the fully second order schemes, Runge-Kutta 2 (RK2), Runge-Kutta 2 with split kernels (SRK2), and dimensional split 2 (DS2) due to the lack of a mixed order version of the SRK2 scheme and the dimensional split mixed order scheme being inferior because of its higher number of domain transpositions per step. If we were to do it for the mixed order schemes as well it would be a separate process of scheme selection identical to what we are doing for the second

order schemes. Blocksize search would not be necessary as the mixed order schemes' kernels are a sub-set of the kernels used for the second order schemes.

To find the fastest scheme we do 5 runs of the ptx version of each of the 3 schemes. We time these runs and select the fastest one to be the chosen scheme for the current GPU. These runs are done at a higher resolution than the micro-benchmarking to reduce the impact of blocksizes that 'wastes' threads outside the domain. This is written to the same csv file as the best blocksizes. During later simulations this is read by the BlocksizePredictor and set as the preferred scheme. To avoid the dimensional split scheme doing recalculations because of large changes in velocities, we set the case for these benchmarks to a sinus wave, which does not produce any large variations in velocities. The dimensional split scheme is already hampered by its need to do smaller timesteps to avoid re-computation. We wish to capture this scheme at its 'best' to see its potential.

The case is, as stated, a sinus wave in X-direction using the following function $w = \sin(2\pi\frac{x}{nx})$, where x is the index of the cell and nx is the number of cells in X-direction. We set $dx = dy = 1$, the bathymetry to $B = 0$, and boundary conditions to wall. For the Q1800M and the GTX435M we used domains with $nx = ny = 1024$, while for the more powerful GTX480 we use $nx = ny = 2048$. The simulation is run to $t = 5$.

Table 4.2 and Table 4.3 show the iterations, time taken, and iterations per second of the test case we use to find the best scheme for the Q1800M and the GTX435M respectively. For both cases the Runge-Kutta 2 with split kernels(SRK2) is the fastest scheme. The same case and resolution (1024x1024) was run on both GPUs.

Scheme	Timesteps	Time taken	Iteration per second
RK2	154	8.40	18.33
SRK2	154	6.78	22.73
DS2	163	9.31	17.50

Table 4.2: This table shows the number of timesteps, total time taken, and iterations per second for the three schemes after blocksize tuning. For this GPU we can see the best scheme is the SRK2 scheme, performing 25% better than the RK2 scheme. The DS2 scheme does not only calculate more timesteps, but also takes longer per timestep.

Scheme	Timesteps	Time taken	Iteration per second
RK2	154	6.42	23.98
SRK2	154	5.96	25.82
DS2	171	8.65	19.76

Table 4.3: Timesteps, time taken and iterations per second of tuned schemes for GTX435M. As with the NVIDIA Q1800M, the SRK2 scheme is the best performing, and the DS2 scheme is vastly inferior.

Table 4.2 contains the same data for the GTX480. For this GPU the

original Runge-Kutta 2 scheme is the fastest, with the SRK2 scheme a close second. Unlike for the other two cases we run a higher resolution (2048x2048) as this card is a lot more powerful and requires a heavier case to run for a few seconds.

Scheme	Timesteps	Time taken	Iteration per second
RK2	154	3.22	47.88
SRK2	154	3.43	44.94
DS2	163	4.82	33.79

Table 4.4: The three schemes number of timesteps, total time taken and iterations per second for a NVIDIA GTX480. Unlike for the two other GPUs the RK2 scheme is the best performing, but is less than 10% better performing that the SRK2 scheme. Also on this GPU we find the DS2 scheme under-performing.

In all of the cases the dimensional split scheme was the slowest. By inspecting a run using the NVIDIA visual profiler we can conclude that the time lost by doing transposition and running more time integration kernels exceeds the time gained by faster flux calculation kernels. If it is possible to make the transpositions in a more effective manner, or improve the time integration kernels, then this scheme might become a contender.

4.4.2 Deciding on early exit

The last part of our auto-tuning system is to dynamically select whether to use early-exit or not. This will vary during a single simulation and therefore has to be selected dynamically at runtime.

We considered a machine-learning approach for this selection of schemes. Each scheme could have been modelled, and execution times or performance predicted. We think it would be too slow to collect the real-time data required to do the predictions. Finding the percentage of the domain that is wet would require loading the wet-map from the GPU and parseing it, or do the parsing using the GPU and load the result. This computation and transfer is likely to cost more than what could have been gained over the dynamic programming scheme we have implemented.

Our dynamic programming scheme is straight forward. Each 100 timesteps is defined as a cycle. For the first timestep of each cycle we run the RK2 scheme with wet-map updating enabled for the last time integration kernel. This is necessary as the previous cycle may have been as scheme without wet-map updating. Having an updated wet-map is important, otherwise when timing the early-exit step, the first flux calculation kernel launched would use an out-dated wet-map and possibly omit calculations of now wet cells. The time integration kernels of the early-exit step will keep updating the wet-map. After this RK2 with wet-map updating, an early-exit step is run and timed. For the third step of the cycle the *preferred scheme*, which was selected by auto-tuning, is run and timed. The fastest one of the *preferred scheme* and early-exit is selected to

be used for the next 97 timesteps. By doing this we ensure that the fastest scheme of the two is always used.

4.5 Impact of complete auto-tuning

In this section we will present the performance of our auto-tuning approach The case run is a variation on the circular dam break described in 3.1.2. In the centre of a domain with size $100m$ x $100m$ there is a cylinder of water, with $R = 6.5m$, which is centred at $x_c = 50m, y_c = 50m$. Initial conditions are

$$h(x,y,0) = \begin{cases} h_{ins} = 10.0m & if (x-x_c)^2 + (y-y_c)^2 \leq R^2 \\ h_{out} = 0m & if (x-x_c)^2 + (y-y_c)^2 > R^2 \end{cases}, \tag{4.1}$$

and $u = v = 0$ for the entire domain. The simulation is run until $t = 4$. At $t = 3$ the entire domain is wet, and so at least the last second of the simulation will be done without early-exit. This way we incorporate both the blocksize tuning of the early-exit scheme and the improvement gained by running a faster scheme when early-exit is not the fastest.

In Figure 4.18 we can see the effect of our entire auto-tuning approach. The four graphs are the tuned *preferred scheme*, labelled 'Tuned', with and without dynamic selection of early-exit enabled, and the original Runge-Kutta 2 scheme with and without dynamic selection of early-exit enabled. We can see an increase in performance by approximately 25% for simulations without early-exit, and approximately 20% for simulations using early-exit. The continued increase in performance with higher resolution for the two early-exit schemes is due to each thread-block becoming a smaller part of the domain, so that the grid of threads blocks can better fit the wet/dry-interface of the domain. This allows a larger percentage of the thread blocks to do early-exit, which increases overall performance.

Significant, although smaller, performance increases can also be seen for the NVIDIA GTX480 in Figure 4.19. For this GPU we have and increase in performance of approximately 20% with early-exit disabled, and approximately 10% with early-exit enabled. Interestingly the tuned versions perform worse than the reference blocksizes for resolutions where the GPU is not fully saturated (roughly $nx = ny = 1000$). This is of no concern, as it is large simulations which are interesting to run on GPUs.

Also for the NVIDIA GTX435M can we see significant increases, see Figure 4.20. For the early-exit simulations we can see increases of approximately 12%. The non-early-exit simulations saw performance gains of roughly 20%. As with the GTX480 we have a decrease in performance for domains too small to saturate the GPU.

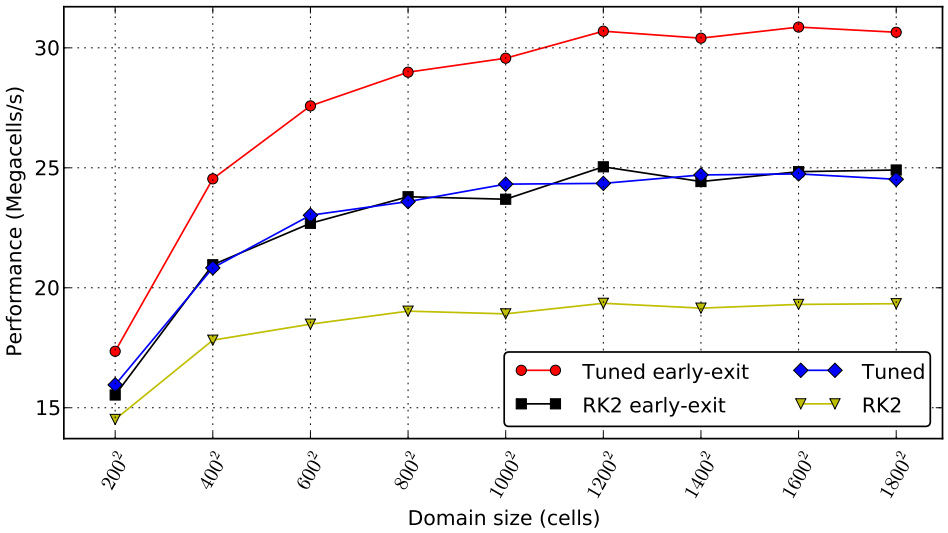


Figure 4.18: The effect of auto-tuning on a NVIDIA Q1800M. The case run is a circular dam break which gradually floods the entire domain. We can see an approximate increase of 5 mega cells per second (20-25%) between tuned and un-tuned runs, both with and without early-exit enabled.

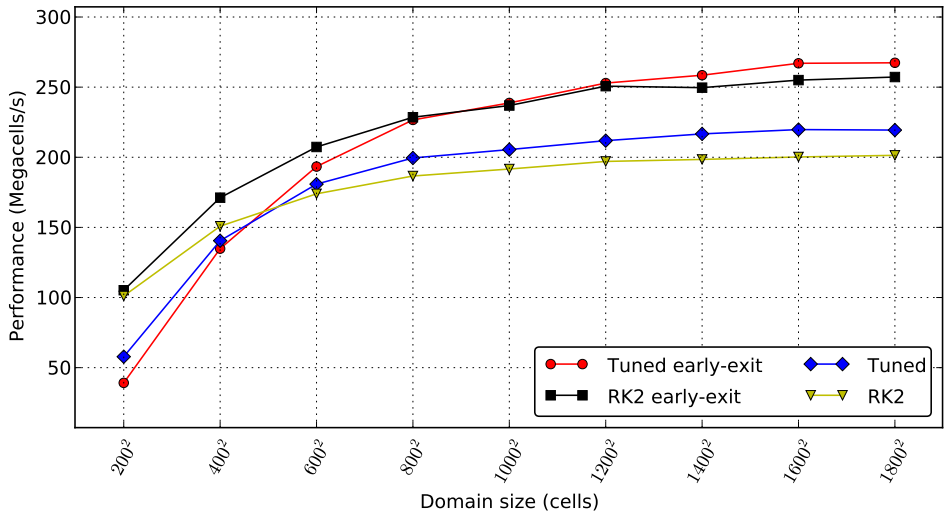


Figure 4.19: The effect of auto-tuning on a NVIDIA GTX480. The case run is a circular dam break which gradually floods the entire domain. We can see a performance increase of approximately 20% for simulations without early-exit, and an increase of approximately 10% for simulation with early-exit.

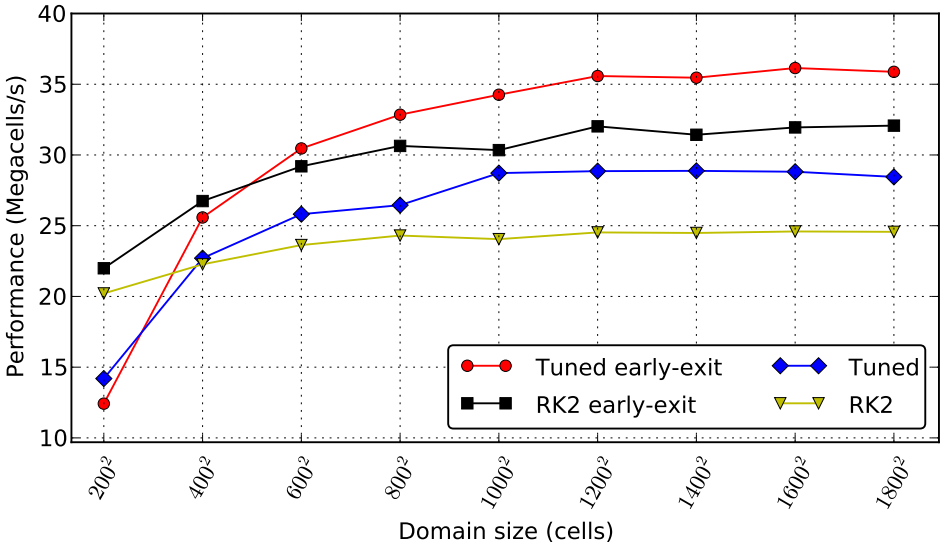


Figure 4.20: The effect of auto-tuning on a NVIDIA GTX435M. The case run is a circular dam break which gradually floods the entire domain. In the figure we can see a performance gain of approximately 20% for non-early-exit simulations, and approximately 12% increase for the early-exit simulations.

Chapter 5

Conclusion

In this thesis we have applied auto-tuning to a shallow water simulator which utilizes NVIDIA GPUs. We have also implemented alternate schemes for approximating the solution of the shallow water equations. These improvements have yielded performance gains in the range of 10-25%.

Two alternative schemes for shallow water simulations were implemented, and are described in chapter 3. Of these two alternate schemes, one proved very efficient on lower-end GPUs, while the other proved inefficient on all three GPUs tested. The under-performing scheme's calculation kernels are significantly faster than that of the original scheme, but the advantage is lost with an increase in the number of kernels launched, and by requiring a transposing of the domain every timestep.

The auto-tuning system we implemented tunes the blocksizes of our 20 most computationally heavy kernels. Before the auto-tuning system was in place, iterating through the search space of a single kernel (through *cmake* and recompilation) took several hours. By compiling and loading kernels during runtime, using CUPTI for micro-benchmarking, and applying pruning of the search space, the time taken by the auto-tuning system was reduced to about 45 minutes total for all kernels. Auto-tuning of the blocksizes yielded performance gains of 10-30% over manually tuned kernels, on three different GPU models. Our system also selects the fastest of the original scheme and the two that was implemented in this thesis. Auto-tuning also removes the need for manual tuning, as was desired.

Our auto-tuning concept is not specific to the kernels in the shallow water simulator used as the target of our auto-tuning. The concept of dynamically compiling and loading kernels, and micro-benchmarking these using CUPTI, should be applicable to any CUDA-program with repeated launches of short duration kernels.

Future research

During our work we encountered topics of interest, which either did not fall within the scope of this thesis, or which time prohibited further research into. These topics could be the subject of future research.

Blocksizes of early-exit In this thesis we optimized the blocksizes of the early exit kernels with emphasis on computation. It may be smaller blocksizes are more beneficial for the early exit kernels, as smaller sizes would be a finer fit to the wet areas of the domain. A finer fit would allow more blocks to opt for early exit. On the downside it would have to use less optimal blocksizes for the blocks where computation is needed, therefore this may be a dynamic parameter varying with the ratio of wet blocks to dry blocks.

Dynamically varying Δt approximation The dimensional split schemes performance varies with the Δt scaling factor, r, selected, and the amount of sudden large changes to water velocity. Dynamically changing r based on the acceleration of the highest velocity water (speed of change in the highest eigenvalues) would cause less timestep recalculations caused by a too large approximation of Δt. This in turn would provide better performance for cases with uneven bathymetry or wet-dry interfaces.

Blocksize prediction using machine-learning It could be possible to eliminate the blocksize tuning by utilizing machine-learning. By training a machine-learning algorithm with relevant information for a variety of GPUs, it may be possible to predict blocksizes across different GPUs without the need for time-consuming micro-benchmarking on every GPU. Bergstra et al. also suggested such an avenue of research[1].

Compilation flags We found that compiling kernels for different target architectures may alter performance of the kernels. It could be interesting to investigate this, and other compiler options, in more detail, as well as incorporating compiler flags in the auto-tuning scheme.

Bibliography

[1] J. Bergstra, N. Pinto and D. Cox. 'Machine learning for predictive auto-tuning with boosted regression trees'. In: *Innovative Parallel Computing (InPar), 2012*. May 2012, pp. 1–9. DOI: 10.1109/InPar.2012.6339587.

[2] A. R. Brodtkorb, M. L. Sætra and M. Altinakar. 'Efficient Shallow Water Simulations on GPUs: Implementation, Visualization, Verification, and Validation'. In: *Computers & Fuids* 55 (2011), pp. 1–12.

[3] R: Caruana and A. Niculescu-Mizil. 'An Empirical Comparison of Supervised Learning Algorithms'. In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML '06. Pittsburgh, Pennsylvania: ACM, 2006, pp. 161–168. ISBN: 1-59593-383-2. DOI: 10.1145/1143844.1143865. URL: http://doi.acm.org/10.1145/1143844.1143865.

[4] K. Datta et al. 'Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures'. In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008, 4:1–4:12. ISBN: 978-1-4244-2835-9. URL: http://dl.acm.org/citation.cfm?id=1413370.1413375.

[5] J. Demmel et al. 'Self-Adapting Linear Algebra Algorithms and Software'. In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 293–312. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840848.

[6] Y. Dotsenko et al. 'Auto-tuning of fast fourier transform on graphics processors'. In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. PPoPP '11. San Antonio, TX, USA: ACM, 2011, pp. 257–266. ISBN: 978-1-4503-0119-0. DOI: 10.1145/1941553.1941589. URL: http://doi.acm.org/10.1145/1941553.1941589.

[7] M. Flynn. 'Some computer organizations and their effectiveness'. In: *Trans. Compute.* C-21 (9 1972), pp. 948–960.

[8] M. Frigo and S. G. Johnson. 'The Design and Implementation of FFTW3'. In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840301.

[9] P. Guo et al. 'A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on GPUs'. In: *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*. TG '11. Salt Lake City, Utah: ACM, 2011, 2:1–2:8. ISBN: 978-1-4503-

0888-5. DOI: 10.1145/2016741.2016744. URL: http://doi.acm.org/10.1145/2016741.2016744.

[10] T. R. Hagen et al. 'How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational device'. In: *Geometrical Modeling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF* (2007), pp. 211–264.

[11] A. Kurganov and D. Levy. 'Central-upwind schemes for the Saint-Venant system'. In: *Mathematical Modelling and Numerical Analysis* 36.3 (2002), pp. 397–425.

[12] A. Kurganov, S. Noelle and G. Petrova. 'Semidiscrete Central-Upwind Schemes for Hyperbolic Conservation Laws and Hamilton–Jacobi Equations'. In: *SIAM Journal on Scientific Computing* 23 (3 2001), pp. 707–740.

[13] A. Kurganov and G. Petrova. 'A Second-Order Well-Balanced Positivity Preserving Central-Upwind Scheme for the Saint-Venant System'. In: *Cummunications in Mathematical Sciences* 5 (2007), pp. 133–160.

[14] R. J. LeVeque. *Finite volume methods for hyperbolic problems*. Vol. 31. Cambridge university press, 2002.

[15] Y. Liu, E.Z. Zhang and X. Shen. 'A cross-input adaptive framework for GPU program optimizations'. In: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. May 2009, pp. 1–10. DOI: 10.1109/IPDPS.2009.5160988.

[16] Y. Li, J. Dongarra and S. Tomov. 'A Note on Auto-tuning GEMM for GPUs'. In: *Proceedings of the 9th International Conference on Computational Science: Part I*. ICCS '09. Baton Rouge, LA: Springer-Verlag, 2009, pp. 884–892. ISBN: 978-3-642-01969-2. DOI: 10.1007/978-3-642-01970-8_89. URL: http://dx.doi.org/10.1007/978-3-642-01970-8_89.

[17] T. G. Mattson, B. A. Sanders and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[18] W. Ma, S. Krishnamoorthy and G. Agrawal. 'Parameterized Micro-benchmarking: An Auto-tuning Approach for Complex Applications'. In: *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 181–182. ISBN: 978-0-7695-4566-0. DOI: 10.1109/PACT.2011.30. URL: http://dx.doi.org/10.1109/PACT.2011.30.

[19] NVIDIA. *CUDA C Programming Guide*. May 2013. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[20] G. Ruetsch and P. Micikevicius. *Optimizing Matrix Transpose in CUDA*. Jan. 2014. URL: http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf.

[21] S. Ryoo et al. 'Program Optimization Space Pruning for a Multithreaded Gpu'. In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '08. Boston, MA, USA: ACM, 2008, pp. 195–204. ISBN: 978-1-59593-978-4. DOI: 10.1145/1356058.1356084. URL: http://doi.acm.org/10.1145/1356058.1356084.

[22] C.-W. Shu. 'Total-Variation-Diminishing Time Discretizations'. In: *SIAM Journal on Scientific and Statistical Computing* 9 (6 1988), pp. 1073–1084.

[23] E. F. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows*. Wiley, 2001.

[24] S. W. Williams. 'Auto-tuning Performance on Multicore Computers'. PhD thesis. EECS Department, University of California, Berkeley, Dec. 2008. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-164.html.

[25] Y. Zhang and F. Mueller. 'Auto-generation and auto-tuning of 3D stencil codes on GPU clusters'. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO '12. San Jose, California: ACM, 2012, pp. 155–164. ISBN: 978-1-4503-1206-6. DOI: 10.1145/2259016.2259037. URL: http://doi.acm.org/10.1145/2259016.2259037.